Programmation Imperative

L1 2021-2022

Projets

Projet 1. ***Système de gestion de location de voiture***

Dans ce projet, vous allez développer un Système de gestion de location de voiture.

Les frais de location de voiture sont calculés en fonction de la catégorie de location, du nombre de voitures louées, de l'emplacement de la voiture et de la durée de la location, des kilomètres parcourus et des jours de location.

Le calcul se fait en fonction de la catégorie comme suit :

Catégorie A (plan de base) :

- Frais de base : 25,00 € par jour.
- Frais kilométriques : 0,15 € par kilomètre.

Catégorie B (plan journalier)

- Frais de base : 35,00 € par jour.
- Frais kilométriques : pas de frais si le nombre moyen de kilomètres parcourus par jour n'excède pas 100. Sinon 0,15 € pour chaque kilomètre au-delà de 100 en moyenne par jour et 0,10 euros pour chaque kilomètre par tranche de 100 kilomètre supplémentaire.

Catégorie C (plan hebdomadaire)

- Frais de base : 120,00 € pour chaque semaine complète ou fraction de semaine
- Frais de kilométrage :
  - Pas de frais si la moyenne des kilomètres parcourus par semaine ne dépasse pas 900.
  - 60,00 € de frais par semaine si le nombre moyen de kilomètres parcourus par semaine dépasse 900 km.
  - 130,00 € de frais par semaine si le nombre moyen de kilomètres dépasse 1500 km et 0,15 € de frais pour chaque kilomètre supplémentaire.

Le tarif final est calculé comme la somme du tarif de base et du tarif kilométrique.

Dans cet exercice, vous allez faire un programme pour stocker les informations relatives à la location de voitures dans des structures appropriées, les traiter, et calculer des informations statistiques sur les frais de location.

Le programme commencera par demander à l'utilisateur quel est le nombre maximum de locations qu'il souhaite obtenir et en fonction de ce nombre il initialisera un tableau dont chaque élément sera un pointeur vers une structure `Rental`. Lors de l'importation des éléments d'une nouvelle location (telle que décrit ci-dessous), il vérifiera s'il reste de l'espace disponible et si c'est le cas la nouvelle location sera insérée.

Les informations suivantes sont demandées et enregistrées dans `Rental` pour chaque location :

- Nombre de jours de location. Représenté par un nombre entier positif.
- Nombre de kilomètres parcourus. Représenté par un nombre réel positif.
- Nom complet du client qui a effectué la location.
- Numéro d'immatriculation de la voiture louée. Correspond à un sept (7) caractères alphanumérique. Les trois premiers caractères sont alphabétiques et les quatre derniers sont numériques. Par exemple, "XNX3456" est un numéro d'enregistrement valide alors que "XN34456", "XNKX3" ne le sont pas.
- Code du plan de facturation, qui est un caractère avec les valeurs possibles "A", "B", "C".

Après avoir déterminé le nombre maximum de locations et les initialisations nécessaires, votre programme s'exécutera de manière itérative. Chaque itération affichera un menu des options à partir duquel l'utilisateur sélectionnera l'option souhaitée :

1. Saisir de nouvelles informations sur la location.
2. Suppression des données de location.

3. Afficher les locations.
4. Arrêt du programme.

Plus de détails :

Saisir de nouvelles informations sur la location.

Tant que l'utilisateur n'a pas dépassé le nombre maximal de locations, il lui est demandé de fournir les informations relatives à la nouvelle location : Nombre de jours de location, nombre de kilomètres parcourus, catégorie de facturation, nombre de kilomètres facturés, numéro d'immatriculation de la voiture louée, nom du client. Pour chaque élément, un contrôle de validité est effectué et si l'élément n'est pas valide, l'utilisateur est invité a le corriger jusqu'à ce qu'un élément valide soit saisi. Une fois que toutes les données ont été saisies, le programme enregistre la nouvelle location.

Suppression des données de location.

L'utilisateur est invité à spécifier une paire de valeurs : le nom et le numéro d'immatriculation de la voiture. La validité des prix saisis est vérifiée, puis le programme recherche et supprime toutes les locations qui ont été effectuées par le client spécifique et pour la voiture spécifique. Le programme libère (free) la mémoire qui correspond aux informations. Lors de la saisie de la paire de valeurs, l'utilisateur peut saisir le caractère alphanumérique "*" dans l'une ou les deux valeurs. Cette valeur spéciale signale que l'utilisateur est intéressé par toute valeur de nom et/ou de numéro d'enregistrement et l'ensemble des locations à supprimer est déterminé en conséquence.

Afficher les locations.

L'utilisateur est invité à spécifier une paire de valeurs : le nom et le numéro d'immatriculation de la voiture. La validité des valeurs saisies est vérifiée, puis le programme recherche et affiche toutes les locations effectuées par ce client et pour cette voiture. Lors de la saisie de la paire de valeurs, l'utilisateur peut saisir le caractère alphanumérique "*" dans l'une ou les deux valeurs. Cette valeur spéciale signale que l'utilisateur est intéressé par l'une des valeurs du nom complet et/ou du numéro d'immatriculation et l'ensemble des locations à afficher est déterminé en conséquence. Après l'affichage des locations, la somme des charges correspondant aux locations affichées est également imprimée.

Arrêt du programme.

Le programme se termine après avoir libéré (free) la mémoire qui était liée dynamiquement.

Projet 2. ***Échecs avec toutes les reines***

Le but de ce projet est de créer un programme permettant de jouer à ce jeu sur le terminal. L'échecs avec toutes les reines s'agit d'une variante des échecs conçue au début des années 2000 par Elliot Rudell.(https ://-ludii.games/details.php ?keyword=All%20Queens%20Ches)

Règles

Chaque joueur dispose de 6 reines d'échecs. Le jeu se joue sur un plateau de 5x5. Dans la position de départ, les reines sont disposées sur des côtés opposés, en alternant blanc-noir-blanc-noir dans chaque case. Les pièces se déplacent comme des reines aux échecs. Plus précisément, la reine peut se déplacer de 1 à 7 cases dans n'importe quelle direction, en haut, en bas, à gauche, à droite ou en diagonale, jusqu'à ce qu'elle atteigne un obstacle. Elle ne peut pas sauter par-dessus les pièces et ne peut capturer qu'une seule pièce par tour. Le premier joueur à **aligner quatre dames** gagne.

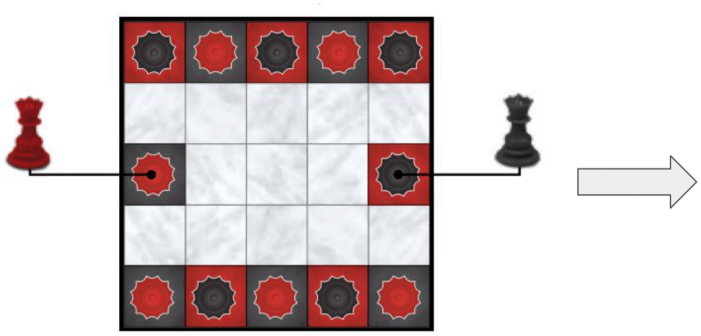
Déplacements autorisés/non autorisés :

1. Les reines peuvent se déplacer de n'importe quel nombre de cases tant qu'elles sont en ligne droite et qu'aucune autre reine ne les gêne.
2. Les reines ne peuvent pas sauter par-dessus une autre.
3. Une reine ne peut pas pousser une autre reine hors d'un espace occupé.

4. Une fois la partie commencée, les espaces couverts par les couronnes sont considérés comme des espaces normaux et peuvent être utilisés par les Reines.

Sauvegarder
L'utilisateur doit pouvoir sauvegarder la partie et la reprendre plus tard ! (pour ce faire, vous devriez utiliser la sauvegarde des positions des dames dans un fichier txt)
Timer
Si un joueur met plus de 7 secondes à jouer(le temps entre le moment où le joueur précédent a choisi son coup et le moment où il a choisi son coup), il perd son tour.
Abandonner
Un joueur peut avoir la possibilité d'abandonner. Dans ce cas, l'autre joueur gagne.
Message gagnant
A la fin d'un jeu, un message annonçant le résultat doit apparaître sur le terminal.
Bonus
Vous pouvez créer une interface graphique à l'aide de la bibliothèque `MLV-2.0.2.` (http ://www-igm.univ-mlv.fr/ boussica/mlv/api/French/html/index.html).

Tâche 1 : Affichage sur le terminal
• `void afficher_chessboard(int chessboard[N][N])` prenant en argument un array2D représentant une grille, et l'affichant joliment sur le terminal.



```
-----------------------------
|   |   |   |   |   |   |
5 | qn | qb | qn | qb | qn |
|   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |
4 |   |   |   |   |   |   |
|   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |
3 | qb |   |   |   | qn |
|   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |
2 |   |   |   |   |   |   |
|   |   |   |   |   |   |
-----------------------------
|   |   |   |   |   |   |
1 | qb | qn | qb | qn | qb |
|   |   |   |   |   |   |
-----------------------------
```

Tâche 2 : Sauvegarder
La deuxière tâche du projet consiste à programmer les deux fonctions suivantes :
• `lire_chessboard(FILE *nom_fichier, int chessboard[N][N])` prenant en argument une chaîne de caractères `nom_fichier` et renvoyant un array 2D décrivant les valeurs des cellules de la grille,
• `void ecrire_chessboard(int chessboard[N][N], FILE *nom_fichier)` prenant en argument une grille sous forme d'array 2D de nombres et un nom de fichier, et sauvegardant la grille fournie dans la fichier indiqué, en respectant le même format.

Tâche 3 : Réalisation du moteur de jeu
Cette tâche du projet consiste à programmer la logique interne du jeu, c'est-à-dire la partie qui permet de

manipuler la structure de données interne représentant l'état de la grille pour simuler le déplacement d'une reine, vérifier si un déplacement d'une reine est autorisé, déterminer si un jeu est fini.

On devra dans cette partie réaliser (au moins) la liste de fonctions suivante. Chacune de ces fonctions reçoit en argument au moins un array2D décrivant le contenu des cellules. Vous pouvez également utiliser une structure `Position` qui contient deux entiers de 0 à 4 qui représentent la position `x` et `y` sur le plateau d'échecs.

Il est bien sûr possible de créer d'autres fonctions, mais les fonctions demandées devront être parfaitement opérationnelles.

• Fonction `int sans_conflit(Position orig, Position fin, int chessboard[N][N])` qui renvoyant 0 si le déplacement d'une reine de `Position orig` vers la `Position fin` est automatisé, et sinon 1.

• Fonction `int winning(int chessboard[N][N])` qui renvoyant 0 si la partie n'est pas terminée, 1 si le joueur blanc a gagné et 2 si le joueur noir a gagné.

Projet 3. ***Sudoku***

Le but de ce projet est de créer un programme permettant de jouer Sudoku sur le terminal.

Règles

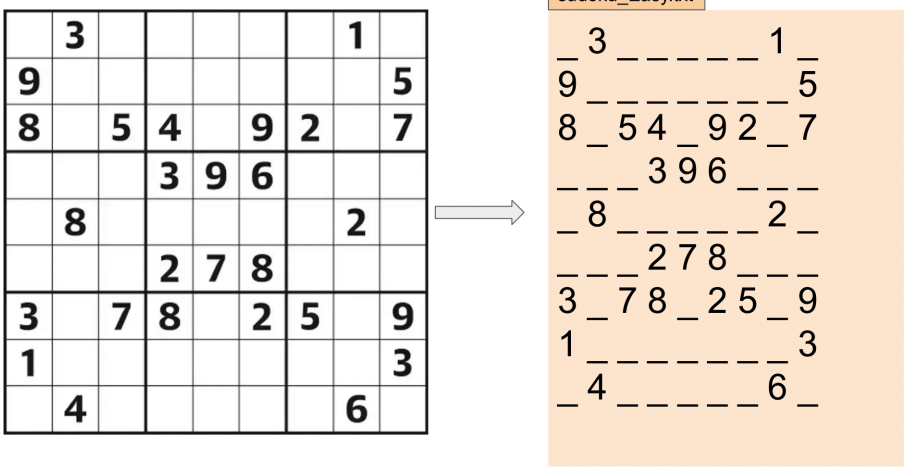
La grille de jeu présentée à droite, à titre d'exemple, est un carré de neuf cases de côté, subdivisé en autant de sous-grilles carrées identiques, appelées « régions ». La règle du jeu générique, donnée en début d'article, se traduit ici simplement : chaque ligne, colonne et région ne doit contenir qu'une seule fois tous les chiffres de un à neuf (1-9). Une règle non écrite mais communément admise veut également qu'une bonne grille de sudoku, une grille valide, ne doit présenter qu'une et une seule solution. Ce n'est pas toujours le cas...

Niveau de difficulté

L'utilisateur doit pouvoir choisir le niveau de difficulté (facile, moyen, difficile). Pour chaque niveau, il doit y avoir (au moins) une grille prédéfinie.

Chargement de la grille

Une grille de Sudoku peut être représentée par un fichier texte comme suit :



La grille est décrite case par case, de la gauche vers la droite puis du haut vers le bas. Chaque caractère du fichier décrit le contenu d'une des cases de la grille : « _ »représente l'absence d'information, et les nombres correspondent aux indices.

Le programme réalisé doit être capable de lire des fichiers écrits dans le format spécifié ci-dessus. Le grille proposées sur la page du projet respecte ce format et vous pourrez donc vous en servir pour tester votre programme.

Le programme doit également détecter d'éventuelles erreurs dans le format d'un fichier (par exemple : présence d'un caractère interdit, lignes de longueurs différentes, etc.), et les signaler par un message d'erreur sur la console.

Sauvegarder

L'utilisateur doit pouvoir sauvegarder la partie et la reprendre plus tard ! (pour ce faire, vous devriez utiliser la sauvegarde des positions des chiffres dans un fichier txt).

Représentation de l'état du jeu

L'état du jeu est représenté par l'information :

• Les valeurs et positions des indices, représentées par un 2D array. Par exemple, les chiffres de la grille donnée en introduction seront représentés comme suit :

```
int sudoku[9][9] = {
    {0, 3, 0, 0, 0, 0, 0, 1, 0},
    {9, 0, 0, 0, 0, 0, 0, 0, 5},
    {8, 0, 5, 4, 0, 9, 2, 0, 7},
    {0, 0, 0, 3, 9, 6, 0, 0, 0},
    {0, 8, 0, 0, 0, 0, 0, 2, 0},
    {0, 0, 0, 2, 7, 8, 0, 0, 0},
    {3, 0, 7, 8, 0, 2, 5, 0, 9},
    {1, 0, 0, 0, 0, 0, 0, 0, 3},
    {0, 4, 0, 0, 0, 0, 0, 6, 0}};
```

Supprimer un cas

Il devrait y avoir une option pour supprimer un chiffre que vous avez fixé plus tôt. Attention à ne pas supprimer un des chiffres qui ont été donnés initialement.

Message gagnant

A la fin d'un jeu, un message annonçant le résultat doit apparaître sur le terminal.

Bonus : Solver automatique

Soit une grille donnée écrire un programme qui trouve la (les) solution (s). Si plusieurs solutions existent, elles doivent toutes être affichées et le programme doit signaler que la règle d'unicité n'est pas respectée.

Tâche 1 : Représentation et chargement des niveaux

La première tâche du projet consiste à programmer les trois fonctions suivantes :

• `lire_grille(FILE *nom_fichier, int grille[N][N])` prenant en argument une chaîne de caractères `nom_fichier` et renvoyant un array 2D décrivant les valeurs des cellules de la grille,

• `void afficher_grille(int grille[N][N])` prenant en argument un array2D représentant une grille, et l'affichant joliment sur le terminal,

• `void ecrire_grille(int grille[N][N], FILE *nom_fichier)` prenant en argument une grille sous forme d'array 2D de nombres et un nom de fichier, et sauvegardant la grille fournie dans la fichier indiqué, en respectant le même format.

Remarques :

Si le fichier fourni n'est pas bien formé (par exemple s'il contient des valeurs inconnues ou des lignes de longueurs différentes), la fonction de lecture pourra provoquer une erreur.

Tâche 2 : Réalisation du moteur de jeu

La seconde tâche du projet consiste à programmer la logique interne du jeu, c'est-à-dire la partie qui permet de manipuler la structure de données interne représentant l'état de la grille pour simuler le remplissage d'une cellule « vide », déterminer si une grille est résolue, et vérifier si les règles ont bien été respectées.

On devra dans cette partie réaliser (au moins) la liste de fonctions suivante. Chacune de ces fonctions reçoit en argument au moins un array2D décrivant le contenu des cellules.

Il est bien sûr possible de créer d'autres fonctions, mais les fonctions demandées devront être parfaitement opérationnelles.

• Fonction `int sans_conflit(int grille[N][N]))` renvoyant 0 si la règle A du jeu est respectée, autrement dit si aucune des cellules visibles de la grille ne contient le même nombre qu'une autre cellule visible située

sur la même ligne ou la même colonne, et 1 sinon.
• Fonction `int sans_conflit_voisines(int grille[N][N])` renvoyant 1 si la règle B du jeu est respectée, autrement dit si aucune cellule visible de la grille ne contient le même nombre qu'une autre cellule visible située sur la même région, et 1 sinon.

Tâche 3 : Recherche de solutions
Cette tâche du projet consiste à implémenter un algorithme de recherche automatique de solution pour le jeu de Sudoku.

- L'**approche naïve** consiste à générer toutes les configurations possibles de chiffres de 1 à 9 pour remplir les cellules vides. Pour chaque position « vide », remplissez la position avec un chiffre de 1 à 9. Après avoir rempli toutes les positions « vides », vérifiez si le tableau est accepté ou pas. Si oui, affichez-la, sinon recommencez pour les autres cas.
- Un autre algorithme permettant de résoudre les grilles de Sudoku est l'algorithme de retour en arrière (**backtracking algorithm**). Vous essayez des chiffres dans les cases vides jusqu'à ce qu'il n'y en ait plus aucun de possible, puis vous revenez en arrière et essayez des chiffres différents dans les cases précédentes.