Programmation Avancée

TP n°8 : Les tableaux

Simon Forest

25 mars 2021

Exercice 1 : Matrices

Dans cet exercice, on implémente une structure pour représenter les matrices. Notez que l'utilisation de l'allocation dynamique ici permet d'avoir des matrices de dimensions variables (avec l'allocation statique, on aurait été obligé d'avoir des dimensions constantes). Cet exercice illustre aussi comment utiliser un double pointeur (comme ici `float**`) pour avoir des tableaux 2-dimensionnels.

1. Introduire une structure `matrix` qui contiendra deux entiers `height` et `width` et un double pointeur `float** values`.
2. Introduire une fonction `void matrix_init(struct matrix* m, int h, int w)` qui initialise une matrice de hauteur `h` et de largeur `w`. Cette fonction devra en particulier :
   - affecter les bonnes valeurs à `m->height` et `m->width`,
   - allouer un tableau de `float*` de taille `h` à `m->values`,
   - pour tout i dans $\{0, \dots, h-1\}$, allouer un tableau de `float` de taille `w` à `m->values[i]` et le remplir de 0.
3. Introduire respectivement une fonction `void matrix_free(struct matrix* m)` qui libère l'espace alloué pour une matrice. Cette fonction devra en particulier :
   - pour tout i dans $\{0, \dots,$ `m->height`$-1\}$, libérer le tableau stocké dans `m->values[i]`,
   - libérer le tableau `m->values`.
4. Écrire une fonction

   ```
   void matrix_add(struct matrix* left, struct matrix* right, struct matrix* res)
   ```

   qui calcule la somme des matrices `left` et `right` et qui met le résultat dans `res`. On supposera que la matrice `res` sera déjà allouée et aura la bonne dimension.
5. Écrire une fonction

   ```
   void matrix_mult(struct matrix* left, struct matrix* right, struct matrix* res)
   ```

   qui calcule la multiplication des matrices `left` et `right` et qui met le résultat dans `res`. On supposera comme avant que `res` est déjà allouée avec les bonnes dimensions.
6. Écrire une fonction `void matrix_print(struct matrix* m)` qui affiche une matrice passée en argument.
7. Tester le code pour vérifier que la somme de

   $$\begin{pmatrix} 1 & 3 \\ 3 & 5 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 4 & 2 \\ 5 & 2 \end{pmatrix}$$

   est la bonne. Vérifier aussi que la multiplication de ces deux matrices est la bonne.

Exercice 2 : Terrains

On représente un terrain rectangulaire de hauteur `h` et de largeur `l` par un tableau 2D avec `h` lignes et `l` colonnes. La case $(0, 0)$ est la première case en haut à gauche et la case $(h-1, l-1)$ est la dernière case en bas à droite. Chaque case du tableau contient un caractère :

- soit `.` si la case en question est libre,
- soit `#` si la case en question contient un obstacle.

Un exemple de tableau décrivant une carte de hauteur 6 et largeur 10 est donné par :

```
##########
#......###
##...#####
##.####..#
##.###...#
##########
```

Ce genre de carte peut être lue dans un fichier. Pour simplifier la lecture, on peut supposer qu'il y aura une première ligne avec les valeurs de `h` et `l`, séparées par un espace. Dans cet exercice, on cherche à lire ce genre de cartes et faire des calculs dessus.

1. Faire un programme qui lit un fichier comme ci-dessus et stocke le contenu de la table dans un tableau 2D. Tester le code en affichant le contenu du tableau après l'avoir lu à partir du fichier contenant l'exemple ci-dessus.

On suppose qu'un humain ne peut être que sur une case « libre » et ne peut se déplacer que d'une case vers le haut, le bas, la gauche ou la droite à la fois. On veut pouvoir connaître l'ensemble des cases où il peut aller en commençant à une certaine case au départ. Pour cela, on propose l'algorithme qui suit. On manipule une liste de positions sous la forme d'une liste chaînée `liste` et contenant initialement la case de départ. Tant que la liste n'est pas vide, on fait les étapes suivantes : D'abord, on récupère et on retire de la liste la première position. Si cette position a déjà été traitée, on recommence en essayant de prendre un nouvel élément de la liste. Sinon, on ajoute à la liste tous les voisins accessibles (au plus 4) et on marque la position actuelle comme « traitée ». Pour marquer une case comme « traitée », on remplacera écrira le caractère `'x'` à la place de `'.'` dans le tableau 2D.

2. Adapter le code des listes chaînées, déjà vues pour les `int`, afin de pouvoir stocker des positions sous la forme de paires d'entiers.

3. Écrire l'algorithme ci-dessus. La case de départ sera récupérée depuis l'entrée du terminal.

4. Tester l'algorithme sur l'exemple ci-dessus en partant de la case à la position $(1, 1)$ (le premier `.` en haut à gauche). En affichant le tableau tel que modifié par l'algorithme, le résultat devrait être :

```
##########
#xxxxxx###
##xxx#####
##x####..#
##x###...#
##########
```