Cours de programmation séquentielle

Matrices

1 Les matrices

1.1 Buts

- Utilisation de pointeurs sur des pointeurs en `C`.
- Utilisation de `git`.
- Utilisation de pointeurs de fonctions.
- Utilisation de fonctions.
- Utilisation de `make`.
- Utilisation de types énumérés.

1.2 Énoncé

Créer une librairie de manipulation de matrices (qui ne sont rien d'autre que des tableaux de tableaux) dans les fichiers, `matrix.h` et `matrix.c`. Vous devez créer un type `struct` nommé `matrix` représentant une matrice de nombres entiers (`int32_t`). Ce type devra contenir les dimensions de la matrice, ainsi que les données contenues dans la matrice. Les données seront représentées sous la forme d'un tableau de tableaux, elles seront donc de type `int32_t **`. Il faut également créer un exécutable qui vous permettra de tester que vos fonctions marchent comme vous le désirez.

1.3 Cahier des charges

Pour ce travail, en plus de la réalisation de la librairie de matrices, vous devez utiliser le logiciel de gestion de version `git`.

1.3.1 Gestion de versions

Pour tout ce travail pratique, vous devez utiliser `git` pour gérer les versions de votre programme. Vous devez utiliser votre compte sur https://gitedu.hesge.ch. Ensuite:

1. Créez le dépôt `matrix`.
2. Clonez les dépôt localement avec la commande `git clone`.
3. Créez un fichier `.gitignore` contenant au moins la ligne suivante `*.o`. Cela permet d'ignorer tous les fichiers `.o` que vous génèrerez à la compilation. Ajoutez ce fichier au dépôt avec la commande `git add .gitignore` et "commitez" le résultat avec la commande `git commit -m "ajout du`

`.gitignore`".[1]

4. Finalement, ajoutez entre autres les fichiers `matrix.h` et `matrix.c`, ainsi que votre `Makefile`, puis commitez le résultat. **N'oubliez pas de compiler régulièrement votre projet et de faire des commits réguliers également.**

1.3.2 Les matrices

Presque toutes les fonctions que vous allez écrire peuvent échouer. Il est donc obligatoire de créer un type énuméré contenant un code d'erreur que retourneront ces fonctions

```
typedef enum _error_code {
    ok, err
} error_code;
```

Pour manipuler des matrices, vous devrez implémenter les fonctions suivantes:

- création d'une nouvelle matrice de `m` lignes et `n` colonnes et allocation de la mémoire

  ```
  error_code matrix_alloc(matrix *mat, int32_t m, int32_t n);
  ```

- allocation et initialisation à une valeur, `val`, d'une nouvelle matrice de `m` lignes et `n` colonnes

  ```
  error_code matrix_init(matrix *mat, int32_t m, int32_t n, int32_t val);
  ```

- libération de la mémoire de la matrice en argument, le pointeur vers les données est mis à `NULL`, le nombre de lignes et de colonnes sont mis à `-1`

  ```
  error_code matrix_destroy(matrix *mat);
  ```

- allocation d'une matrice, et initialisation de ses valeurs à partir d'un tableau de taille `s = m*n`

  ```
  error_code matrix_init_from_array(matrix *mat, int32_t m, int32_t n,
                                    int32_t data[], int32_t s);
  ```

- création du clone d'une matrice, la nouvelle matrice est une copie de la matrice d'origine (il faut réallouer la mémoire)

  ```
  error_code matrix_clone(matrix *cloned, matrix mat);
  ```

- création de la matrice transposée d'une matrice, la nouvelle matrice est une copie de la matrice originale ou les lignes et les colonnes sont échangées

  ```
  error_code matrix_transpose(matrix *transposed,
                              matrix mat);
  ```

- affichage d'une matrice (très utile pour le débogage)

  ```
  error_code matrix_print(matrix mat);
  ```

[1] Typiquement dans un fichier `.gitignore` on ajoute tous les fichiers binaires du dépôt pour éviter de les ajouter par erreur et de les versionner.

- test de l'égalité de deux matrices

  ```
  bool matrix_is_equal(matrix mat1, matrix mat2);
  ```

- récupération de l'élément `[ix][iy]` de la matrice de façon sûre (vérification des dépassements de capacité par exemple) et copie dans `elem`

  ```
  error_code matrix_get(int32_t *elem, matrix mat,
                        int32_t ix, int32_t iy);
  ```

- modification d'un élément `[ix][iy]` de la matrice de façon sûre (vérification des dépassements de capacité par exemple)

  ```
  error_code matrix_set(matrix mat, int32_t ix, int32_t iy,
                        int32_t elem);
  ```

  Le type matrice sera défini en `C` de la manière suivante

```
typedef struct _matrix {
    int32_t m, n;
    int32_t ** data;
} matrix;
```

Bien que cela ne soit pas optimal d'un point de vue de la performance, vous devez allouer `data` comme étant d'abord un tableau de `m` pointeur d'entier, puis chaque case de `data`, contiendra un tableau de `n` entiers.

En utilisant les assertions créez un programme `matrix_test.c` testant chacune des fonctionnalités implémentée plus haut. Puis avec une cible `test` dans votre `Makefile` compilez et exécutez les tests pour savoir si votre librairie fonctionne correctement. Pensez à utiliser les `assert()` pour cette partie.

Écrire également un programme de test `matrix_compute.c` permettant de vérifier (en affichant les résultats à l'écran) que votre programme marche.

1.3.3 Exercice supplémentaire

Comme exercice supplémentaire vous pouvez implémenter les fonctions suivantes nécessitant des pointeurs de fonctions (voir ce lien et ce lien):

- appliquer une fonction sur chaque élément d'une matrice "en place" (*in place* en bon français)

  ```
  error_code matrix_map_ip(matrix mat, void (*foo)(int32_t *));
  ```

  par exemple la fonction `foo` pourrait être la fonction multipliant par deux un nombre. Ainsi on multiplierait aisément tous les éléments de la matrice par deux.

- appliquer une fonction sur chaque élément d'une matrice et enregistrer le résultat dans une nouvelle matrice

  ```
  error_code matrix_map(matrix *mapped, matrix mat,
                        void (*foo)(int32_t *));
  ```

1.4 Indications

- Pensez à compiler souvent: le compilateur est votre ami.
- Évitez d'écrire toutes les fonctions en une fois sans tester si les fonctionnalités marchent comme vous le souhaitez.
- Pensez à utiliser les warnings et les *sanitizers*, cela peut vous sauver d'erreurs terribles et très difficiles à découvrir

  ```
  -Wall -Wextra -pedantic -fsanitize=address -fsanitize=leak
  ```

- Lorsque vous voyez apparaître des warnings corrigez-les immédiatement et pas "quand vous aurez fini". Il arrive régulièrement qu'un warning vous indique une erreur de logique dans votre code.

1.5 Rapide introduction/rappel aux matrices

Une matrice est un **tableau de nombres**, a un nombre de lignes noté, m, et un nombre de colonnes noté n. Pour simplifier, on dit que c'est une matrice $m \times n$. La matrice $\underline{\underline{A}}$ ci-dessous, a 3 lignes et 4 colonnes

$$\underline{\underline{A}} = \begin{pmatrix} 2 & 1 & -1 & -2 \\ 3 & 1 & 1 & 3 \\ 1 & 4 & -1 & -1 \end{pmatrix}, \tag{1}$$

on dit donc que c'est une matrice 3×4.

Chaque élément d'une matrice peut être accédé par une paire d'indices, i, j (i étant le numéro de la ligne, j le numéro de la colonne), et est noté par A_{ij}. Dans le cas ci-dessus, l'élément $A_{14} = -2$.

On peut définir la matrice *transposée* de la matrice $\underline{\underline{A}}$, notée $\underline{\underline{A}}^{\mathrm{T}}$, comme la matrice obtenue en inversant tous les indices de $\underline{\underline{A}}$. On a que $A^{\mathrm{T}}_{ij} = A_{ji}$. Si $\underline{\underline{A}}$ est une matrice $m \times n$, alors $\underline{\underline{A}}^{\mathrm{T}}$ est une matrice de taille $n \times m$.

Exemple (Transposée)
Pour la matrice

$$\underline{\underline{A}} = \begin{pmatrix} 2 & 1 & -1 & -2 \\ 3 & 1 & 1 & 3 \\ 1 & 4 & -1 & -1 \end{pmatrix}, \tag{2}$$

la matrice transposée $\underline{\underline{A}}^{\mathrm{T}}$ sera

$$\underline{\underline{A}}^{\mathrm{T}} = \begin{pmatrix} 2 & 3 & 1 \\ 1 & 1 & 4 \\ -1 & 1 & -1 \\ -2 & 3 & -1 \end{pmatrix}. \tag{3}$$

Finalement, pour que deux matrices soient égales, il faut que tous leurs éléments soient égaux et que leurs tailles soient les mêmes évidemment.