Programmation C

Feuille n° 2 – TP machine

17 septembre 2026

Matrices carrées

Cette feuille est issue du premier TP noté de l'année 2024. Le barème d'origine a été laissé pour information. Le TP noté était prévu pour 1h20. Comme il a eu lieu exactement un mois plus tard dans l'année universitaire, il est normal d'y consacrer plus de temps pour le réaliser : essayez, par exemple, de le terminer en $2 \times$ 1h20. Si vous ne l'avez pas terminé en fin de séance d'aujourd'hui, terminez-le chez vous pour bien vous préparer.

Avant de commencer, vous devez avoir lu les fiches techniques sur `le shell`, `make` et `VS Code`.

1 Objectif du TP

On veut programmer des algorithmes sur des matrices carrées de nombres entiers, c'est-à-dire des tableaux carrés d'entiers. On appellera **taille** d'une telle matrice son nombre de lignes (ou de colonnes). Une matrice de taille n contient donc n^2 entiers. Par exemple, la matrice suivante est de taille 4 et contient 16 entiers.

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

L'objectif du TP est de coder :

- Deux implémentations simples mais différentes d'opérations de base sur les matrices carrées.
- Des fonctions permettant de créer des matrices et de les afficher en les parcourant selon un ordre précis.

Chacun de ces objectifs est détaillé plus bas, et dans les fichiers d'en-tête `matrix.h` et `print.h`.

2 Ce que vous devez télécharger

1. Téléchargez et décompressez l'archive `tp1.tgz`, descendez dans le répertoire créé `tp1`, lancez `VS Code` grâce aux 4 commandes la fiche `VS Code`.
2. Éditez les fichiers `matrix1.c`, `matrix2.c`, `print.c`, `main.c` dans `src/` et `version.h` dans `include/`.

2.1 Ce qui est fourni

2.1.1 Fichiers sources C et fichiers d'en-tête

Les fichiers `.c` sont dans `src/`. Les fichiers `.h` sont dans `include/`. Le fichier `main.c` contient des tests. Vous pouvez le modifier. Des tests y sont déjà écrits. Les résultats attendus sont obtenus en lançant le programme avec les `#if` des fichiers `matrix1.c`, `matrix2.c` et `print.c` positionnés à `#if 0` (comme c'est le cas au départ). Les fichiers à éditer sont **uniquement** `main.c`, `matrix1.c`, `matrix2.c`, `print.c` et `version.h`. Les autres fichiers (en-têtes `matrix.h` et `print.h`, fichier `Makefile`) **ne doivent pas être modifiés**.

2.1.2 Le fichier `Makefile`

Le fichier `Makefile` fournit beaucoup d'avertissements pour vous aider à déboguer. Les avertissements sont traités comme des erreurs[1] Votre code doit se compiler sans avertissement. Le `Makefile` construit l'exécutable `matrix`, que vous pouvez exécuter sous le shell par la commande `./matrix`. Vous devez coder deux versions d'une bibliothèque, la première dans `matrix1.c`, la seconde dans `matrix2.c`. Le `Makefile` fourni permet de créer l'exécutable avec l'une ou l'autre des bibliothèques.

Comment éditer et compiler ?

Si la macro `VERSION` de `include/version.h` vaut `1`, éditez le fichier `matrix1.c` : c'est lui qui sera compilé. Si `VERSION` vaut `2`, c'est `matrix2.c` qui sera compilé. Une fois choisie la version de la bibliothèque que vous voulez coder, il faut refaire le fichier `compile_commands.json` (voir cadre suivant). Ensuite, vous pourrez compiler avec :

```
$ make
```

Travaillez sur une fonction à la fois. Les fonctions entourées par `#if 0 ... #endif` ne sont pas utilisées : c'est une correction qui est utilisée à la place. Par conséquent, quand vous travaillez sur une fonction, changez son `#if 0 ... #endif` en `#if 1 ... #endif`. C'est alors votre version qui sera utilisée. Lorsque vous avez fini une fonction, vous pouvez rebasculer `#if 1` à `#if 0`, et passer à la fonction suivante.

Important : changement de version de la bibliothèque

Lorsque vous changez de bibliothèque, c'est-à-dire que vous passez au codage de `matrix2.c` après avoir codé `matrix1.c`, ou inversement que vous revenez à `matrix1.c` après avoir codé `matrix2.c`, il faut une commande particulière de compilation. Pour changer de version, changez d'abord la macro `VERSION` du fichier `include/version.h` à la version voulue, `1` ou `2`. Puis, juste après ce changement, recompilez une fois avec :

```
$ make -B compile_commands.json
```

Comment nettoyer ?

Pour nettoyer les fichiers produits par la compilation :

```
$ make clean
```

Cette commande supprime le fichier `matrix`, ainsi que les fichiers `.o` et `.d` créés par `gcc`.

3 Fonctions à écrire

Il y a trois parties à écrire, chacune dans un fichier : `matrix1.c`, `matrix2.c` et `print.c` (voir Sections 3.1, 3.2 et 3.3). Les deux premiers fichiers, `matrix1.c` et `matrix2.c`, fournissent deux implémentations différentes d'une même bibliothèque. Leurs fonctions sont courtes (parfois une seule instruction). Il est possible de commencer les fonctions de `print.c` dès qu'un des fichiers `matrix1.c` ou `matrix2.c` est complet.

Les descriptions précises des fonctions à écrire sont détaillées dans les fichiers `matrix.h` et `print.h`.

Important : Ne perdez **pas** de temps à tester si les arguments passés à votre programme sont corrects. (cette tâche est laissée à la charge de l'utilisateur des fonctions).

1. Sauf les arguments inutilisés (car au départ, la fonction que vous écrirez sera vide, donc n'utilisera pas ses arguments).

3.1 Première implémentation de la micro-bibliothèque : `matrix1.c` (3 points)

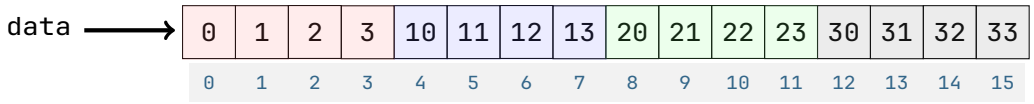
Cette implémentation est à réaliser dans le fichier `matrix1.c`. Une matrice est codée par le type suivant :

```c
typedef struct matrix {
  int size;
  int *data;
} matrix;
```

Ici, `size` est la taille de la matrice à représenter (c'est son nombre de lignes, ou de colonnes car ces nombres sont égaux puisque la matrice est carrée). Le principe est que `data` permet de stocker les éléments de la matrice, les uns à la suite des autres en mémoire. Une fois alloué, `data` devra donc pointer sur la première case d'une zone de `(size * size)` entiers, contenant les lignes de la matrice, lues de haut en bas. Par exemple, pour la matrice suivante, `size` vaut 4.

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \end{pmatrix}$$

Pour la représenter, il faudra allouer, en plus de la structure `matrix`, le tableau suivant pour le champ `data` :

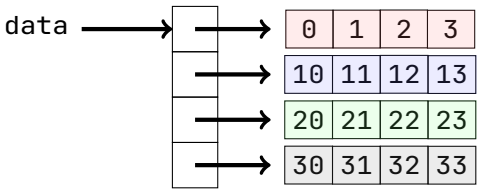


3.2 Seconde implémentation de la micro-bibliothèque : `matrix2.c` (3 points)

Cette implémentation est à réaliser dans le fichier `matrix2.c`. Une matrice est codée par le type suivant :

```c
typedef struct matrix {
  int size;
  int **data;
} matrix;
```

À nouveau, `size` est la taille de la matrice à représenter. Remarquez que le type de `data` est **différent** de celui de la première implémentation : il pointe sur la première case d'un tableau de `size` pointeurs sur des entiers. Le i-ème de ces pointeurs pointe sur la première case de la i-ème ligne de la matrice (en commençant à $i = 0$). Par exemple, pour allouer la mémoire représentant la même matrice que ci-dessus, il faudra allouer, en plus de la structure `matrix`, **plusieurs** tableaux. De même, la désallocation est un peu plus compliquée qu'avec l'implémentation précédente.



3.3 Fonctions de test de la bibliothèque : `print.c` (14 points)

On demande enfin d'écrire des fonctions de test des bibliothèques. Le travail demandé est décrit dans `print.h`. Notez que les fonctions des deux bibliothèques ont les mêmes noms, et le fichier d'en-tête est le même, `matrix.h`. Cependant, suivant la valeur de la macro `VERSION` définie dans `include/version.h` (soit 1, soit 2), l'une seulement des bibliothèques sera utilisée. Les fonctions de `print.c` doivent fonctionner **avec l'une comme l'autre des bibliothèques**. Autrement dit, vous ne devez pas utiliser l'implémentation, mais les fonctions de bibliothèque. Par exemple, dans `print.c`, pour affecter 42 en ligne 2, colonne 5 d'une matrice `M`, on utilisera `set_matrix_value(M, 2, 5, 42);`. Essayer de modifier directement `M->...` échouera, car `print.c` ne connaît pas l'implémentation des matrices.