Allocation de tableaux en C

Objectifs pédagogiques:

- Allouer/libérer de la mémoire avec malloc/free;
- Manipuler des pointeurs/tableaux;
- Créer de nouveaux types de données avec le mot-clé `struct`;
- Utiliser en pratique les outils gdb et valgrind.

L'objectif de ce TP est de manipuler les fonctions malloc et free, et de vous entrainer à debugger un programme s'arrêtant sur SEGFAULT. L'excuse du jour sera la création d'une bibliothèque permettant de créer et manipuler des vecteurs et des matrices.

★ **Principe du TP.** Veuillez télécharger l'archive :
`https://mquinson.frama.io/ensr-arcsys1/TP6-allocations.tgz`

Vous trouverez dans cette archive plusieurs programmes à compléter. Vous pouvez compiler et tester tous les exercices de la séance avec la commande `make test`

Chaque test correspond à un programme (le nom du programme s'affiche après ECHEC ou SUCCES) que vous pouvez exécuter indépendemment pour déterminer vos erreurs. Le test <`toto`> est passé si votre programme `toto` affiche exactement la même chose que ce qui se trouve dans le fichier `toto.result`. N'hésitez pas à consulter le contenu des fichiers fournis pour comprendre le fonctionnement de l'ensemble. Si votre programme vous semble fonctionner, mais que la suite de tests vous indique le contraire, comparez le contenu du fichier `toto.output` produit par votre programme au fichier `toto.result`, qui est la sortie attendue par la suite de tests. Utilisez pour cela la commande `diff -u toto.result toto.output` qui affiche les différences ligne par ligne. Il est bien entendu possible de modifier `toto.result` pour faire en sorte que les tests ne détectent plus le problème, mais ce n'est pas l'objectif ;)

Pour comprendre vos segfaults, utilisez valgrind en exécutant: `valgrind --leak-check=full ./toto`

★ **Exercice 1: Vecteurs.** Nous souhaitons réaliser les fonctions nécessaires à la gestion d'un type vecteur.

▷ **Question 1:** Complétez ***vector.c*** qui contient toutes les fonctions de gestion de vecteur que nous voulons implémenter. Le fichier ***vector.h*** contient toutes les informations nécessaires : description des fonctions et déclaration du type vecteur. Une fois toutes les fonctions complétées, tapez `make t11-vector` pour compiler, `./t11-vector` pour exécuter le programme de test et make test pour tester si le résultat du programme de test est correct (s'il affiche la même chose que ce qui se trouve dans ***t11-vector.result***).

▷ **Question 2:** Ajoutez un test dans la fonction d'accès, et renvoyez NULL si l'indice demandé est en dehors des bornes de l'objet concerné. Testez votre modification avec le programme `./t12-vector-bound`

▷ **Question 3:** On a supposé à la question précédente qu'accéder à une case inexistante est une erreur. Parfois, on préfère faire en sorte que le vecteur grandisse automatiquement pour créer à la demande les cases inexistantes. Comme il est difficile de prédire le comportement souhaité par l'utilisateur de la bibliothèque, on va configurer le comportement avec le champ `dynamic` de la structure vecteur. S'il vaut FAUX (`dynamic == 0`), on utilisera le comportement de la question précédente: un accès à une case inexistante renvoi NULL. Si le champ vaut VRAI (`dynamic != 0`), il faut agrandir le tableau (avec `realloc()`) quand l'indice dépasse la borne.

Testez votre modification avec le programme `./t13-vector-realloc`.

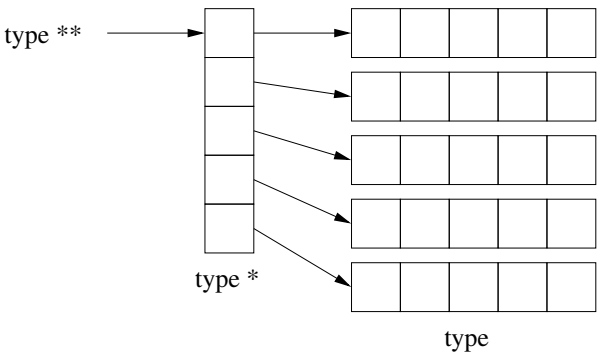
★ **Exercice 2: Matrices.** On veut implémenter un type `matrix_t`, représentant les matrices ainsi:

Pour allouer une telle structure, on alloue d'abord un tableau de pointeurs, dont la taille correspond au nombre de lignes souhaité. Ensuite, chacune des cases de ce premier tableau est un pointeur vers un autre tableau alloué séparément pour représenter l'une des lignes. Le code correspondant est donné page suivante.

On accède ensuite aux cases du tableaux comme on le ferait pour un tableau alloué statiquement: avec l'opérateur `[ ]` pour chacune des dimensions:

Initialisation des éléments

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 10; j++)
        tableau[i][j] = 0;
```



Structure d'un tableau bidimensionnel dynamique

Allocation d'un tableau de 10*10 doubles

```
double** tableau;

tableau = malloc(sizeof(double*) * 10);
for (int i = 0; i < 10; i++) {
    tableau[i] = malloc(sizeof(double) * 10);
}
```

Idem avec gestion des erreurs

```
double** tableau;

tableau = malloc(sizeof(double *) * 10);
if (tableau == NULL) { // malloc signale une erreur
    abort(); // tue le programme en signalant un bug interne
}
for (int i = 0; i < 10; i++) {
    tableau[i] = malloc(sizeof(double) * 10);
    if (tableau[i] == NULL) {
        abort();
    }
}
```

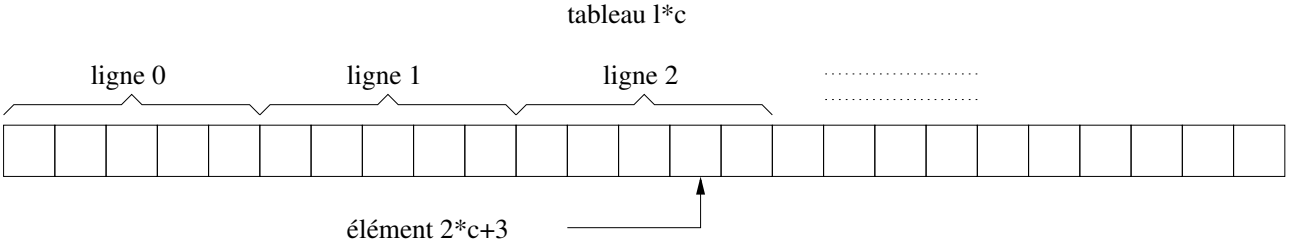
▷ **Question 1:** Complétez le fichier ***matrix.c*** qui contient toutes les fonctions de gestion de matrice que nous voulons implémenter. Le fichier ***matrix.h*** contient toutes les informations nécessaires : description des fonctions et déclaration du type `matrix_t`. Une fois les fonctions complétées, tapez `make t21-matrix` pour compiler, `./make t21-matrix` pour exécuter le programme de test et `make test` pour tester si le résultat du programme de test est correct.

▷ **Question 2:** Modifiez votre fonction d'accès pour renvoyer NULL si l'on accède à une case qui n'existe pas, et testez votre travail avec `./t22-matrix-bounds`.

▷ **Question 3:** Modifiez maintenant votre fonction d'accès pour qu'elle augmente la taille de la matrice lorsqu'on accède à une case qui n'existe pas encore, à condition que la matrice soit dynamique. Testez votre travail avec `./t23-matrix-realloc`

★ **Exercice 3: Matrices linéaires (optionnel)**

En pratique, les matrices sont très rarement représentées en mémoire comme nous avons fait à l'exercice précédent. Au lieu de cela, on utilise un seul gros tableau unidimensionel. Les lignes de la matrice sont alors stockées les unes après les autres, de façon contiguë. La fonction d'accès traduit un couple d'indices de la matrice en un unique indice dans le tableau. On économise ainsi une indirection, ainsi que le coût des structures de gestion mémoire associées à chaque `malloc` par le système. Les tableaux multidimensionels statiques sont automatiquement stockés sous cette forme en mémoire.



▷ **Question 1:** Implémentez le type `matlin_t` (pour *mat*rice *lin*éaire) dans le fichier ***matlin.c***. Comme d'habitude, la documentation se trouve dans ***matlin.h*** et vous pouvez tester votre travail avec `./t31-matlin`.

▷ **Question 2:** Étendez votre fonction d'accès afin de retourner NULL quand on accède à une case inexistante, et testez votre travail avec `./t32-matlin-bounds`.

▷ **Question 3:** Étendez à nouveau votre fonction d'accès afin d'agrandir la matrice lorsqu'on accède à une case inexistante d'une matrice dynamique, puis testez votre travail avec `./t33-matlin-realloc`.

★ **Exercice 4: Opérations mémoire (optionnel)** Complétez le fichier ***memory.c*** afin d'implémenter les fonctions de manipulation mémoire suivantes:

- `my_memcpy`: copie d'une zone en mémoire de la même manière que `memcpy` (cf. man);
- `my_memmove`: copie d'une zone en mémoire avec recouvrement possible, c'est-à-dire que la zone à écrire peut chevaucher partiellement la zone à lire (cf. `man memmove`). Il est parfois nécessaire de commencer par recopier les octets de la fin;
- `is_little_endian`: renvoie vrai si l'architecture cible utilise la convention *little endian*;
- `reverse_endianess`: renvoie la valeur passée en argument avec ses octets inversé.

Vérifiez votre travail avec `./t41-memory-cpy` `./t42-memory-move` et `./t43-memory-endianess`.