TP5 : Allocation de tableaux en C

CSH : Initiation au C et au shell

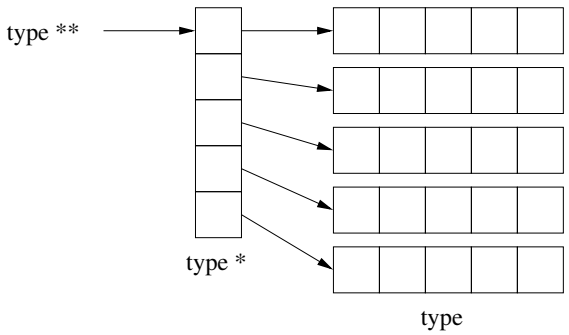
L'objectif de ce TP est d'appliquer les méthodes de debug vues au TP précédent pour mettre au point des programmes manipulant la mémoire. L'excuse du jour sera la création d'une bibliothèque permettant de créer et manipuler des vecteurs et des matrices. À la fin de ce TP vous saurez :

— Créer un type de données particulier ;
— Utiliser en pratique les outils de mise au point de programme C gdb et valgrind.

★ **Introduction.** En C, l'allocation dynamique d'un tableau multidimensionnel se fait en deux étapes :

— allocation d'un tableau de pointeurs dont la taille correspond au nombre de lignes souhaité
— allocation de chacune des lignes en utilisant le tableau de pointeurs précédent pour stocker leur adresse

Ensuite, l'accès au tableau se fait comme d'habitude en C, en utilisant l'opérateur [] pour préciser l'indice dans chacune des dimensions. La seule difference avec un tableau alloué de manière statique est le type des objets manipulés : dans le cas dynamique ce sera un tableau de pointeurs, dans le cas statique ce sera un tableau de tableaux (les tableaux sont réellement stockés les uns après les autres en mémoire).



Structure d'un tableau bidimensionnel dynamique

Exemple: allocation d'un tableau de 10*10 doubles (sans aucune vérification)

```c
double **tableau;

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

Idem avec libération en cas d'erreur

```c
double **tableau;

tableau = (double **) malloc(sizeof(double *)*10);
if (tableau != NULL) {
  i=0;
  erreur=0;
  while ((i<10) && !erreur) {
    tableau[i] = (double *) malloc(sizeof(double)*10);
    if (tableau == NULL)
      erreur = 1;
    else
      i++;
  }
  if (erreur) {
    while (i) {
      i--;
      free(tableau[i]);
    }
  }
}
```

Initialisation des éléments

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

Vous trouverez dans le dépot 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 avant ECHEC ou SUCCES) que vous pouvez exécuter independemment 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. Pour cela, vous pouvez utiliser 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 ;)

★ **Exercice 1: Vecteurs**

Nous souhaitons réaliser les fonctions nécessaires à la gestion d'un type vecteur. Complétez ***vecteur.c*** qui contient toutes les fonctions de gestion de vecteur que nous voulons implémenter. Le fichier ***vecteur.h***

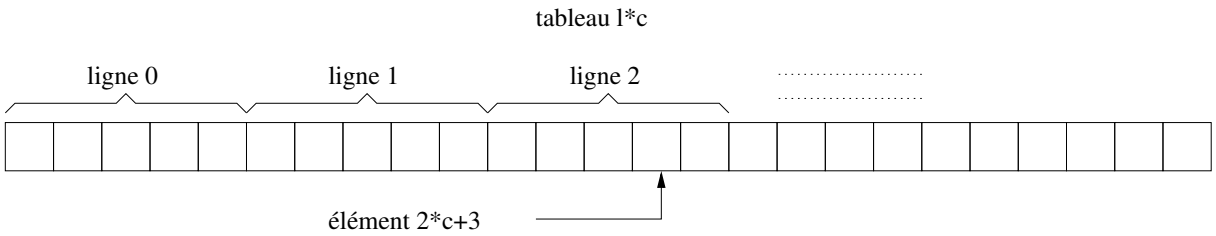
contient toutes les informations nécéssaires : description des fonctions et déclaration du type vecteur. Une fois les fonctions complétées, tapez `make vecteur_testbase` pour compiler, `./vecteur_testbase` pour executer le programme de test et make test pour tester si le résultat du programme de test est correct (s'il affiche la même chose que ce qui se trouve dans ***vecteur_testbase.result***).

★ **Exercice 2: Matrices**

Nous souhaitons réaliser les fonctions nécessaires à la gestion d'un type matrice, où les matrices sont représentées par un tableau bidimensionel dynamique. Complétez le fichier ***matrice.c*** qui contient toutes les fonctions de gestion de matrice que nous voulons implémenter. Le fichier ***matrice.h*** contient toutes les informations nécessaires : description des fonctions et déclaration du type matrice. Une fois les fonctions complétées, tapez `make matrice_testbase` pour compiler, `./matrice_testbase` pour exécuter le programme de test et `make test` pour tester si le résultat du programme de test est correct.

★ **Exercice 3: Matrices linéaires**

Nous souhaitons réaliser les fonctions nécessaires à la gestion d'un type matrice, où les matrices sont représentées par un seul tableau unidimensionel. Dans ce cas, les lignes de la matrice sont stockées les unes après les autres et la fonction d'accès va devoir faire la traduction d'un couple d'indices vers un unique indice dans le tableau (l'avantage est que l'on économise une indirection ainsi que le coût des structures de gestion mémoire associées à chaque `malloc` par le système). La figure suivante décrit l'organisation de nos matrices en mémoire :



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

★ **Exercice 4: Vérifier les bornes**

Reprenez les trois exercices précédents en ajoutant un test dans la fonction d'accès permettant de renvoyer NULL si les indices demandés sont en dehors des bornes de l'objet concerné. Les fichiers à completer sont ***vecteur_verif.c***, ***matrice_verif.c*** et ***matrice_lineaire_verif.c***. Les commandes sont analogues aux exercices précédents.

★ **Exercice 5: Réallocation dynamique**

Reprenez les trois premiers exercices en ajoutant une réallocation dynamique des objets dans la fonction d'accès. Le comportement de cette fonction doit alors être le suivant :

- si un des indices d'accès est négatif retourner `NULL` ;
- si les indices sont dans les bornes de l'objet retourner le pointeur d'accès au bon élément ;
- si un des indices dépasse les bornes de l'objet tenter de réallouer plus de mémoire et renvoyer le pointeur d'accès au bon élément si la réallocation réussit et `NULL` sinon.

Pour la réallocation, on pourra utiliser au choix un `malloc` d'un bloc plus gros suivi d'une copie ou bien l'appel système `realloc` dont on obtient la description avec la commande `man realloc`.

Complétez ***vecteur_dynamique.c***, ***matrice_dynamique.c*** et ***matrice_li neaire_dynamique.c***.

★ **Exercice 6: Opérations mémoire.** Implémentez 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. (cf. `man memmove`) ;
- `is_little_endian` : renvoie vrai si l'architecture cible utilise la convention little endian pour la représentation des entiers en mémoire ;
- `reverse_endianess` : renvoie la valeur passée en argument avec ses octets inversé.

Le fichier à completer est ***memory_operations.c***.