TP6 : Allocation de tableaux en C

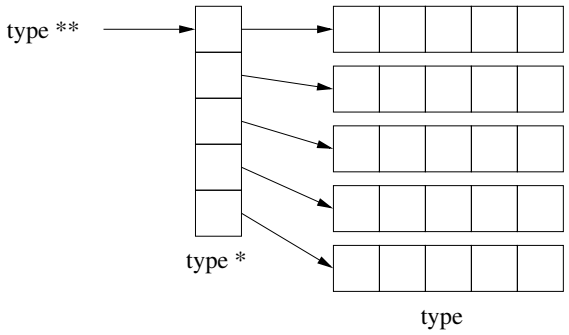
L'objectif de ce TP est d'appliquer les méthodes de debug vues au TP précédent pour mettre au point des programmes manipulant la mémoire. L'excuse du jour sera la création d'une bibliothèque permettant de créer et manipuler des vecteurs et des matrices. À la fin de ce TP vous saurez :

- 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)

```
1  double **tableau;
2
3  tableau = (double **) malloc(sizeof(double *)*10);
4  for (i=0; i<10; i++)
5      tableau[i] = (double *) malloc(sizeof(double)*10);
```

Idem avec libération en cas d'erreur

```
1  double **tableau;
2
3  tableau = (double **) malloc(sizeof(double *)*10);
4  if (tableau != NULL) {
5    i=0;
6    erreur=0;
7    while ((i<10) && !erreur) {
8      tableau[i] = (double *) malloc(sizeof(double)*10);
9      if (tableau == NULL)
10       erreur = 1;
11     else
12       i++;
13   }
14   if (erreur) {
15     while (i) {
16       i--;
17       free(tableau[i]);
18     }
19   }
20 }
```

Initialisation des éléments

```
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**

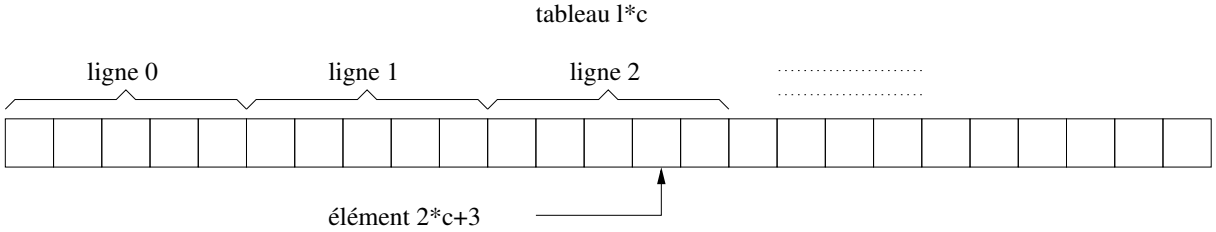
Nous souhaitons réaliser les fonctions nécessaires à la gestion d'un type vecteur. Complétez ***vecteur.c*** qui contient toutes les fonctions de gestion de vecteur que nous voulons implémenter. Le fichier ***vecteur.h*** contient toutes les informations nécéssaires : description des fonctions et déclaration 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_te` 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***.

★ **Exercice 7. Dice Poker** Ce jeu est une variation du Poker qui utilise des dés au lieu de cartes. Vous allez maintenant implémenter un programme permettant de jouer à ce jeu contre l'ordinateur.

Le jeu consiste en plusieurs manches. Dans chaque manche, l'ordinateur et l'humain lancent 5 dés. Certaines combinaisons de valeur de dés permettent de l'emporter sur d'autres. La liste suivante énumère les combinaisons de la plus "puissante" à la moins "puissante" :

(7) *Full :* les cinq dés ont la même valeur ;
(6) *Quatre d'un coup :* 4 des 5 dés ont la même valeur, et le dernier est différent ;
(5) *House :* 3 dés ont la même valeur, tandis que les deux autre ont une autre valeur, identique ;
(4) *Straight :* Cinq valeurs consécutives (même dans le désordre) ;
(3) *Trois d'un coup :* 3 dés ont la même valeur, tandis que les deux autres sont différents ;
(2) *Deux paires :* Deux paires de dés ont la même valeur ;
(1) *Paire :* 2 dés ont la même valeur, et les autres sont différents ;
(0) *Rien*

▷ **Question 1.** Observez le code du programme ***dice_poker.c*** fourni. N'hésitez pas à consulter la page man des fonctions utilisées que vous ne connaissez pas. Compilez-le et exécutez-le plusieurs fois pour vérifier son bon fonctionnement.

▷ **Question 2.** Ajoutez une fonction `int manche()` à ce programme, qui crée deux tableaux de 5 entiers tirés aléatoirement, et les affiche sous la forme suivante :

```
Moi : 3 5 4 4 5
Vous: 3 3 6 1 2
```

Vous utiliserez une fonction `void affiche(char *name, int hand[5])` pour afficher chacune des lignes.

▷ **Question 3.** Réalisez une fonction `int identifie(int hand[5])` permettant de reconnaître le type de main obtenu. Par exemple, si toutes les valeurs sont différentes, elle renverra la valeur 0, tandis qu'elle renverra 7 si toutes les valeurs sont identiques (*cf.* le nombre entre parenthèses au début de chaque item de la liste des combinaisons).

Pour cela, il faudra calculer la fréquence de chaque valeur de la main (dans un tableau temporaire). Ainsi, pour la main {3,5,4,4,5} la table de fréquence est {0,0,1,2,2,0} (il y a zéro fois la valeur 1, zéro fois la valeur 2, une fois la valeur 3, et ainsi de suite). Le cas du "Straight" se traite à part.

▷ **Question 4.** Modifiez la fonction `affiche()` pour qu'elle affiche le type de main obtenu par chacun.

▷ **Question 5.** Modifiez `manche()` pour qu'elle affiche le gagnant de la manche et renvoi -1 si l'ordinateur gagne, 0 sur match nul et 1 si l'humain gagne.

```
Moi : 3 5 4 4 5 (deux paires)
Vous: 3 3 6 1 2 (paire)
JE GAGNE !
```

▷ **Question 6.** Modifiez la fonction principale pour jouer 5 manches, compter les points acquis à chaque manche et afficher le résultat final.

▷ **Question 7.** Modifiez votre programme pour autoriser les jeux avec un nombre arbitraire de dés, spécifié en ligne de commande. La fonction d'évaluation doit naturellement être modifiée (on peut laisser les Straight de coté dans un premier temps). Il faut pour cela consulter non seulement la table des fréquences de valeur, mais également la table des fréquences de la table des fréquences.

Par exemple, {4,6,2,4,3,6,6,6,3} a la table de fréquence suivante : {0,1,2,2,0,5} et la table de fréquence des fréquences suivante : {2,1,2,0,0,1,0,0,0,0}. Il y a deux valeurs qui ne sont pas représentées (le 1 et le 5), une valeur unique (le 2), deux paires (3 et 4), et un groupe de cinq valeurs identiques (les 6).

Valeur	1	2	3	4	5	6
Fréquence	0	1	2	2	0	5

(a) Table des fréquences

Fréquence	0	1	2	3	4	5	6	7	8	9
Nombre de valeurs ayant cette fréquence	2	1	2	0	0	1	0	0	0	0

(b) Table des fréquences de fréquence

Un *full* d'une main de 9 dés a la table de fréquence des fréquences suivantes : {5,0,0,0,0,0,0,0,0,1} (5 valeurs ne sont pas représentées du tout tandis qu'une valeur est représentée neuf fois).

On peut considérer cette table comme les coefficients d'un polynôme, dont il convient d'ignorer les deux premiers termes. La main du premier exemple correspond ainsi à $2 \times X^2 + X^5$ (on ne retient que la présence de deux paires et d'un groupe de cinq valeurs identiques) tandis que le full est X^9.

Nous voila prêts à modifier la fonction `affiche()` pour donner une forme textuelle de la combinaison obtenue. On utilisera les abréviations grecques ou latines pour les valeurs supérieures à 4. La main précédente est "2 paires, 1 penta" tandis que $X^4 + X^6$ est "1 carré, 1 hexa" et le full "1 nona".

Pour la valeur de la combinaison, on évaluera le polynôme pour $X = 2$ (attention, 4 paires sont plus fortes qu'un triple) ou $X = 10$ (attention aux débordements de capacité, on peut obtenir des valeurs de main négatives si on s'y prend mal).

▷ **Question 8.** Faites en sorte de permettre à l'utilisateur d'indiquer les dés qu'il souhaite relancer (comme on redemande une carte au Poker) en indiquant 5 booléens après chaque lancé. Dans l'exemple suivant, les caractères soulignés sont tapés par l'utilisateur qui demande à relancer les 3 derniers dés.

```
Moi : 3 5 4 4 5 (deux paires)
Vous: 3 3 6 1 2 (paire)
Que voulez vous faire? 0 0 1 1 1
Vous: 3 3 3 4 4 (house)
VOUS GAGNEZ !
```

▷ **Question 9.** Limitez les cas d'égalité en faisant en sorte qu'entre deux Fulls, le gagnant soit celui dont les dés marquent la plus grande valeur. Faites de même pour les combinaisons "N d'un coup". Pour départager deux "House", on regarde d'abord la valeur des triplettes avant de regarder la valeur des paires si les triplettes sont équivalentes. On départage les "Deux paires" de façon similaire. Un "Six Straight" (2-6) est plus fort qu'un "Five Straight" (1-5).

▷ **Question 10.** *(mini projet)* Dotez l'ordinateur d'une stratégie pour lui aussi relancer certain de ses dés. Cette décision sera prise par le biais d'une fonction `int *rejoue(int *hand, int hand_size)`.

▷ **Question 11.** Implémentez plusieurs fonctions de ce type (tout rejouer, rien rejouer, rejouer un dé une fois sur deux, ainsi que des "vraies" stratégies), et faites jouer les AI les unes contre les autres sur un grand nombre de parties pour déterminer la meilleure approche. Faites jouer votre AI contre celle de votre voisin.