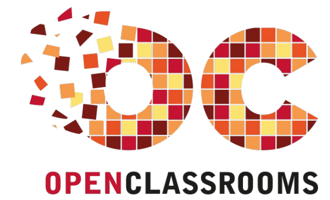
Tableaux, pointeurs et allocation dynamique

Par uknow

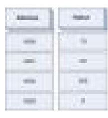


www.openclassrooms.com

Licence Creative Commons 6 2.0
Dernière mise à jour le 25/10/2012

Sommaire

Sommaire ... 2
Les tableaux de pointeurs ... 1
Tableaux, pointeurs et allocation dynamique ... 3
Les pointeurs (rappel) ... 3
Définition ... 3
Déclaration ... 3
Utilisations ... 4
Les tableaux unidimensionnels ... 6
Définition ... 6
Déclaration (syntaxe) ... 7
Utilisation ... 7
Allocation dynamique d'un tableau à une dimension ... 14
Les tableaux de tableaux ("tableaux à plusieurs dimensions") ... 17
Déclaration ... 18
Utilisation ... 18
Les pointeurs sur tableaux ... 22
Les tableaux de pointeurs ... 23
Allocation dynamique ... 24
Fonctions : mettre un tableau en argument ... 36
Fonctions : retourner un tableau ... 41
Exercices ... 42
Pointeurs ... 42
Tableaux unidimensionnels ... 44
Tableaux multidimensionnels ... 45
Partager ... 46



Tableaux, pointeurs et allocation dynamique

Par uknow

Mise à jour : 25/10/2012
Difficulté : Difficile Durée d'étude : 3 jours

(cc) BY-NC-SA

Bien que les tableaux et les pointeurs soient souvent confondus, il s'agit de deux choses différentes.

Nous allons aborder ce sujet dans le but d'utiliser correctement chacun de ces types (pointeur et tableau) et de comprendre la différence entre ces deux notions.
Nous allons également voir certaines notions approfondies dans les manipulations des tableaux (les initialisations, les déclarations, les passages en paramètre pour les fonctions).

Sans plus tarder attaquons le vif du sujet en vous souhaitant une bonne lecture.

Conseils pour une bonne lecture :

- Une bonne concentration.
- Un projet "test" ouvert, pour coder au fur et à mesure que vous lisez.
- Ne pas passer à l'étape suivante sans comprendre l'étape précédente.
- Faites des exercices à chaque fin de partie du tutoriel.
- Ne lisez pas le tutoriel en entier d'un seul coup.
- Suspendez la lecture si vous vous sentez largués. Et reprenez la lecture plus tard.

Sommaire du tutoriel :

- Les pointeurs (rappel)
- Les tableaux unidimensionnels
- Les tableaux de tableaux ("tableaux à plusieurs dimensions")
- Exercices

Les pointeurs (rappel)

Définition

A la déclaration d'une variable, un emplacement lui est accordé dans la mémoire. Cet emplacement possède une adresse. Cette adresse peut être stockée dans une variable de type pointeur.

Pour résumer un pointeur est une variable qui contient l'adresse mémoire d'une autre variable.

Déclaration

L'opérateur de déclaration de pointeur est l'astérisque '*', et il est caractérisé par le type de variable sur laquelle il va pointer.
Ainsi pour déclarer un pointeur on doit respecter la syntaxe suivante : **type** `*nom_du_pointeur;`

Si par exemple on souhaite déclarer un pointeur sur une variable de type int, on ferait comme ceci :

Code : C

```c
int * ptrint;
```

Utilisations

La règle à retenir est la suivante :

- ***Un pointeur doit toujours être initialisé avant utilisation.***

Initialisation

Pourquoi il faut toujours initialiser les pointeurs ?

A la déclaration d'une variable quelconque, sa valeur ne peut pas être déterminée. Elle peut valoir "n'importe quoi". Les pointeurs étant des variables aussi, alors à la déclaration ils valent n'importe quoi . Cette valeur se réfère donc à un emplacement mémoire dont on ignore sa signification. Et qui ne nous est pas alloué, donc inutilisable.

Et comment initialiser un pointeur ?

L'initialisation peut avoir trois formes :

- Avec la valeur `NULL` :

  Code : C

  ```c
  int * ptrint = NULL;
  ```

  Le pointeur `NULL` souvent sous la forme `(void*)0`, est une macro déclarée dans *stddef.h*. Elle est utilisée pour des opérations exclusivement sur le type pointeur (affectation, comparaison...).

- Avec l'adresse d'une de nos variables déclarées.

  Code : C

  ```c
  int variable;
  int * ptrint = &variable;
  ```

  Rappelez-vous de l'opérateur '&', qui permet d'avoir l'adresse mémoire d'un objet. Il s'utilise toujours accompagné du nom de l'objet que l'on souhaite connaitre son adresse mémoire.
  Avec cette initialisation, on est sûr que l'espace mémoire que notre pointeur indique, nous est alloué.

- En faisant une allocation dynamique (que nous allons voir plus loin dans ce tutoriel).

  Code : C

  ```c
  int * ptrint = malloc(10);
  ```

Donc si vous déclarez un pointeur, ayez à l'esprit qu'il faudra l'initialiser tôt ou tard avec l'une des trois méthodes présentées ci-dessus, en fonction de ce que vous souhaitez en faire.
Autrement il y a de fortes chances pour que votre programme plante avec l'erreur "SEGFAULT".

Accéder à l'adresse pointée

L'astérisque '*', vous l'avez reconnue . C'est ce même opérateur qui est utilisé pour déclarer une variable de type pointeur et pour accéder à l'emplacement indiqué par notre pointeur.
Dans l'apprentissage de cette partie, les débutants confondent toujours quand est-ce qu'il faut utiliser l'étoile et quand est-ce qu'il ne faut pas l'utiliser.
Gardez à l'esprit que l'étoile, à la déclaration d'une variable de type pointeur, ne signifie pas qu'on accède à l'emplacement qu'il pointe, mais sert juste à dire au compilateur qu'il s'agit d'une déclaration de pointeur.
Mis à part ce cas là, toutes les autres utilisations de l'opérateur '*' suivi du nom d'un pointeur, signifient que c'est de l'emplacement pointé qu'il s'agit.

Exemple :

Code : C

```c
int * ptrint;         //Déclaration du pointeur
int variable;         //Déclaration d'une variable

ptrint = &variable;   //Initialisation de notre pointeur

*ptrint = 10;         //On inscrit 10 à l'espace pointé par notre
pointeur (en l'occurrence la variable).
```

Argument de fonctions : passage par valeur

Lors de l'appel d'une fonction, on lui donne des valeurs sous forme de paramètres. Il existe donc le type de passages qu'on appelle par valeur, qui consiste à passer une copie de notre valeur à la fonction. Ainsi la fonction ne manipulera que cette copie de notre valeur, et tous les changements qui y seront apportés, ne seront pas pris en compte ailleurs.

Exemple :

Code : C

```c
void ma_fonction(int n)   //Fonction appelée
{
      n = 10;
}

int main (void)                             //Fonction appelante
{
      int variable = 123;

      ma_fonction(variable);
      printf("La valeur de variable est %d\n",variable);

      return 0;
}
```

Le résultat de ce code sera donc :

Code : Console

```
La valeur de variable est 123
```

Pourtant on a bien modifié la valeur envoyée à la fonction 'ma_fonction'. C'est ce qu'on appelle un passage par valeur .

Argument de fonctions : passage par adresse

Ce type de passage donnera un résultat différent de celui présenté précédemment. Il consiste à envoyer l'adresse mémoire de notre variable et non pas sa valeur. Ainsi on a accès à l'emplacement même de cette variable en mémoire. Donc les changements qu'on apportera à cet espace mémoire, seront des changements qu'on aura apportés à notre variable directement. Si je reprends le même exemple, ceci donnerait :

Code : C

```
void ma_fonction(int * ptrn)
{
      *ptrn = 10;     //On inscrit la valeur 10 à l'emplacement 
mémoire indiqué par ptrn
}

int main (void)
{
      int variable = 123;

      ma_fonction( &variable );                          //On 
appelle la fonction
      printf("La valeur de variable est %d\n",variable);

      return 0;
}
```

Et le résultat est :

Code : Console

```
La valeur de variable est 10
```

C'est ce qu'on appel un passage par adresse .

Argument de fonctions : passage par référence

Le passage par référence tel que nous pouvons le voir dans d'autres langages (C++ par exemple) ne peut pas être réalisé en C; mais il peut être implémenté par utilisation du "passage par adresse".
Ce passage par adresse est un passage par valeur quelque part, car on transmet une copie de l'adresse de la variable à la fonction. Donc ne soyez pas choqué si on vous dit quelque part qu'il n'y a pas de passage par référence en C.

Les tableaux unidimensionnels

Définition

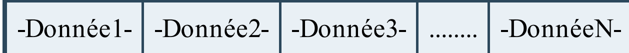
"*Un tableau est une suite contigüe de données de même type dans la mémoire.*"

Beaucoup de gros mots dans cette phrase n'est-ce pas ?

Si nous les regardons de plus près :

Une suite contigüe :

Une suite contigüe signifie un ensemble d'éléments disposés les uns à la suite des autres sans être intercalés.



-Donnée1-	-Donnée2-	-Donnée3-	....	-DonnéeN-

Données de même type :

Ceci signifie que toutes les données qu'on va retrouver dans notre tableau vont être du même type .

Si nous prenons l'exemple ci-dessus, donnée1, donnée2 jusqu'à la donnéeN vont être exactement du même type.

Déclaration (syntaxe)

La déclaration d'un type tableau en C s'obtient par utilisation des crochets '[' ']', et en précisant le type de données qu'il y aura dedans ainsi que le nombre.

Le nombre de ces éléments sera appelé la taille de notre tableau, et leur type le type de notre tableau.

La déclaration sera donc sous la forme : **type** `nom_du_tableau` [**taille**];

Voici un exemple de déclaration d'un tableau de 10 entiers :

Code : C

```c
int tableau[10];
```

Utilisation

Initialisation

Il existe certaines expressions **réservées à l'initialisation lors de la déclaration d'un tableau**.

Code : C

```c
int tableau[3];
tableau = ....;
```

Erreur très courante à l'utilisation des chaines de caractères.

Ces expressions sont :

- L'utilisation des accolades {} :
  Code : C

  ```c
  int tableau[4] = {15,2,14,23};
  ```

  Les cases de ce tableau qu'on vient de déclarer seront donc initialisées respectivement avec les valeurs 15, 2, 14 et 23. Vous remarquerez que le nombre de valeurs correspond à la taille de notre tableau (à savoir 4 éléments). Il est possible de n'initialiser qu'un certain nombre de cases dans ce tableau ainsi :
  Code : C

```c
int tableau[4] = {15,2};
```

Ainsi nous aurons initialisé que les deux premières cases. Vous vous demandez peut être ce que deviennent les autres cases , elles sont toutes initialisées avec des 0 (zéros).

En conséquence une initialisation à zéro de la première case uniquement, initialiserait tout notre tableau à zéro :

Code : C

```c
int tableau[4] = {0};
```

- L'utilisation des accolades {} avec un tableau dont la taille n'est pas spécifiée :

Code : C

```c
int tableau[] = {15,2,14,23};
```

Deux choses très importantes à noter dans ce type de déclarations :
1_ Le tableau sera alloué pour contenir le nombre d'éléments présents entre les accolades (en l'occurrence 4).
2_ Le tableau sera initialisé avec ces éléments.

Par conséquent, il ne faut pas mélanger cette déclaration avec l'initialisation type {0} :

Code : C

```c
int tableau[] = {0};
```

Car cela ne va allouer qu'une seule case. Erreur très courante dans le cas des chaines de caractères.

Que faire pour n'initialiser que certaines cases dans le tableau ?

Seul le standard C99 permet de faire ce genre d'initialisations, contrairement au C90.

Code : C

```c
int t[10] = {[indice]=5,6};
```

indice ici doit être remplacé par une valeur constante indiquant l'indice qu'on souhaite initialiser avec la valeur 5, il est évidemment obligatoire que cet indice soit inférieur à la taille du tableau.

Pour se familiariser avec ce type d'initialisations prenons quelques exemples :

Exemple 1 :

Code : C

```c
int t[10] = {2,3,[4]=5,6,9};
```

résultat :

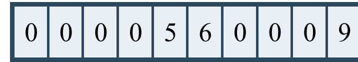
2	3	0	0	5	6	9	0	0	0

Exemple 2 :

Code : C

```c
int t[10] = {[4]=5,6,[9]=9};
```

résultat :

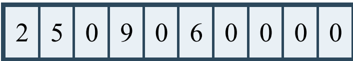


Exemple 3:

Code : C

```c
int t[10] = {2,3,[1]=5,[5]=6,[3]=9};
```

résultat :



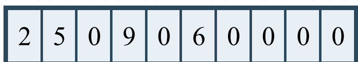
Ce que je ne vous ai pas encore dit, est que si l'indice spécifié correspond à une case déjà initialisée, alors sa valeur sera écrasée par la nouvelle valeur (par le 5 dans cet exemple dans un premier temps) et toute les autres cases restantes à sa "droite" seront initialisées à 0; le même phénomène est constaté avec [3]=9, la case d'indice 3 sera initialisée à 9 alors que toutes les cases restantes seront mises à 0.
Je vous propose de suivre les étapes de plus près :
Etape 1 : Les deux premières cases sont initialisées à 2 et 3 (les autres ont pour le moment des valeurs indéterminées 'x').

2	3	x	x	x	x	x	x	x	x

Etape 2 : la case d'indice 1 est initialisée à 5, et les cases restantes à droite sont initialisées à 0 jusqu'au prochain indice ([5]).

2	5	0	0	0	6	x	x	x	x

Etape 3 : la case d'indice 3 est initialisée à 9, et les cases restantes à droite sont toutes initialisées à 0.



Exemple 4:

Code : C

```c
int t[10] = {1,[10]=5,6,[14]=9};
```

résultat :
Erreur de compilation, en effet, il est interdit de mettre un indicateur d'indice qui dépasse la taille de notre tableau.

Exemple 5:

Code : C

```c
int t[] = {1,[10]=5,6,[14]=9};
```

résultat : Nous avons déclaré un tableau de type "incomplet"; en d'autres mots, un tableau dont la taille n'a pas encore été donnée. Ce tableau prendra donc la taille de l'initialisateur, et dans ce cas de figure, le tableau sera de taille 15 car le plus grand indice est le 14 (les indices seront donc entre 0 et 14 donc une taille de 15 cases).

- L'utilisation des doubles-cotes `""` :

Cette initialisation est réservée aux déclarations de tableaux de type `char` appelés aussi chaines de caractères, elle est

très identique aux déclarations présentées ci-dessus.

Code : C

```
char tableau[4] = "ABC";
```

Ici le tableau sera initialisé avec les valeurs 'A', 'B', 'C' et '\0'.

Code : C

```
char tableau[4] = "ABCD";
```

Ceci va initialiser les 4 premières cases du tableau avec les données 'A', 'B', 'C' et 'D', le '\0' ne sera pas placé en fin de la chaine par défaut de taille. Donc ceci est une source de bogues très courants quand vous essayez de manipuler ce tableau.
Veillez toujours à compter le '\0' en plus des caractères d'initialisation.

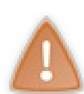
Code : C

```
char tableau[] = "ABC";
```

Ici le tableau sera alloué pour contenir les données 'A', 'B', 'C' et '\0'. Donc aura une taille de 4 cases.

Code : C

```
char tableau[] = "";
```



Ceci va allouer un tableau d'une seule case, donc prenez en garde lors de vos manipulations de ce tableau.

Parcourir un tableau

Le parcours d'un tableau s'effectue par l'utilisation des mêmes crochets '[]' utilisés à la déclaration de notre tableau, et en indiquant le rang de la case à laquelle on souhaite accéder. Par exemple pour écrire le nombre 12 dans la première case on ferait :

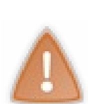
Code : C

```
tableau[0] = 12;
```

Le '0' ici est appelé indice de la première case.

Code : C

```
int tableau[4];
tableau[4] = 15;
```



Pour un tableau de 4 cases, les indices commencent à 0 et vont jusqu'à 3 inclus, donc la case d'indice 4 n'existe pas!

Maintenant qu'on sait que les indices d'un tableau commencent à 0 et vont jusqu'à taille - 1 (taille étant le nombre de cases de notre tableau), on va voir maintenant comment parcourir ce tableau à l'aide d'une boucle.

Pour ceci, il nous faudrait une variable qui jouera le rôle de l'indice, qu'on déclarera comme ceci :

Code : C

```
int i;
```

Ensuite, par utilisation d'une boucle, on va parcourir les cases du tableau pour y mettre des zéros :

Code : C

```
int i;

for( i=0 ; i < taille ; i++)  //Attention la condition est : i < 
taille !
{
      tableau[i] = 0;
}
```

'typedef' et le type tableau

Comme vous le savez peut être, le mot-clef typedef permet de définir (ou redéfinir) des types. C'est-à-dire, que l'on peut changer l'identificateur d'un objet par utilisation de typedef.

Je m'explique :

Code : C

```
typedef int my_int;
```

Me permettrait d'utiliser my_int comme étant un type (en l'occurrence `int`). C'est très utile pour alléger le code. L'exemple d'application classique sont les structures.

La syntaxe afin de définir un type tableau à l'aide du mot-clef typedef est :

typedef type_du_tableau `nom_du_tableau` [**taille_du_tableau**];

Exemple : Si l'on veut définir un type de tableau de 4 entiers, on ferait ainsi :

Code : C

```
typedef int tab4 [4];

int main(void){
      tab4 tableauDe4 = {1,2,3,4};

      return 0;
}
```

Il est possible que l'on ait besoin de déclarer un type tableau sans taille, on ferait donc ainsi :

Code : C

```
typedef int tab [];
```

```c
int main(void){
     tab tableauDe4 = {1,2,3,4};

     return 0;
}
```

Qui est strictement équivalent à faire comme nous l'avons vu dans la partie "initialisation" à savoir :

Code : C

```c
int tableauDe4[] = {1,2,3,4};
```

Passer un tableau en argument à une fonction

Il existe quatre façons peu différentes de le faire, et ceci en respect à ce qu'on a vu dans les parties déclaration et initialisation :

- Utiliser la déclaration d'un tableau :

  Si l'on dispose d'un tableau de taille 'N' qu'on souhaite passer en argument à une de nos fonctions, on ferait donc comme ceci :

  Code : C

  ```c
  void ma_fonction(int tableau[N]);
  ```

  Et si l'on ne connait pas à priori la taille de notre tableau (ou que l'on ne souhaite pas utiliser d'indication sur la taille), on peut faire comme ceci :

  Code : C

  ```c
  void ma_fonction(int tableau[]);
  ```

  Remarque : `int tableau[N]` , `int tableau[]` , et `int * tableau` sont strictement équivalentes seulement pour les paramètres formels d'une fonction.

- Utiliser un pointeur

  On peut utiliser un pointeur pour réceptionner un tableau envoyé en argument, en déclarant l'argument ainsi :
  Code : C

  ```c
  void ma_fonction(int * tableau);
  ```

- Utiliser un `typedef`

  Comme on l'a vu, on pourrait utiliser un `typedef` pour déclarer un argument de type tableau, on ferait donc ainsi :
  Code : C

  ```c
  typedef int tabN[N];
  ```

```c
void ma_fonction(tabN tableau);
```

Ou sans utiliser de taille :

Code : C

```c
typedef int tab[];

void ma_fonction(tab tableau);
```

Retourner un tableau

La syntaxe est la suivante :

Code : C

```c
int (ma_fonction(void))[4];
```

Mais si vous essayez de le faire le compilateur vous rejettera . Ce qui est normal, car le langage C ne le permet pas, pour la raison suivante :

- **Hors cas d'utilisation du mot-clef `static`**, la déclaration d'un tableau est faite de manière automatique, ce qui fait qu'il a une portée locale. Donc le retourner à une autre fonction, impliquerait l'utilisation d'un espace mémoire déjà détruit.

  Code : C

  ```c
  int (ma_fonction(void))[5]{    //Ceci génère une erreur de syntaxe
        int tab[5] = {0};
        return tab;                 //Retour du tableau
  }
  ```

  ou

  Code : C

  ```c
  int  (* fonction(void))[5]{   //ceci est correct syntaxiquement mais aboutirait à un bogue
        int tab[5] = {0};
        return &tab;                            //Retour d'un pointeur sur le tableau
  }
  ```

  ou

  Code : C

  ```c
  int * fonction(void){   //ceci est correct syntaxiquement mais aboutirait à un bogue
        int tab[5] = {0};
  ```

```
		return tab;						//Retour d'un
	pointeur sur le premier élément du tableau
	}
```

ou

Code : C

```
	int  * fonction(void){  //ceci est correct syntaxiquement
	mais aboutirait à un bogue
		int tab[5] = {0};
		return &(tab[0]);					//Retour
	d'un pointeur sur le premier élément du tableau
	}
```

On est donc obligé d'allouer dynamiquement notre tableau et le retourner sous la forme d'un pointeur.

Ou d'utiliser une structure , dans laquelle on mettra un tableau, et le fait de retourner cette structure, ne retournera qu'une copie de cette dernière. Donc cette méthode est correcte, et bien qu'elle soit pratique, mais relève du "bourrin" dans le codage (c'est pourquoi je ne vous encouragerai pas à l'utiliser).

Exemple :

Code : C

```
typedef struct {
	int tableau[5];
}STableauDe5;

STableauDe5 ma_fonction(void){
	STableauDe5 tab = {{0}};

	return tab;
}
```

Ceci est valable à la seule condition de réceptionner ce retour dans une variable de type STableaDe5 :

Code : C

```
int main (void){
	STableauDe5 tab;

	tab = ma_fonction();
	return 0;
}
```

Quant à la méthode d'allocation dynamique nous la verrons plus tard.

Allocation dynamique d'un tableau à une dimension

Une allocation dynamique consiste à demander au système d'exploitation de nous allouer un espace d'une taille donnée dans la mémoire (dans le tas).

Par abus de langage on utilise les termes "allocation dynamique de tableau". Il est donc important de dissocier une allocation d'une taille en mémoire et le type tableau qu'on a vu précédemment.

La fonction malloc

La fonction malloc est la plus populaire des fonctions d'allocation, j'en rappelle le prototype :

Code : C

```c
void *malloc (size_t size);
```

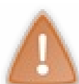
- **size** : la taille en "byte" (octet par abus de langage) de l'espace mémoire à allouer.
- **La valeur retournée** : la fonction malloc retourne un pointeur sur le premier "byte" de l'espace alloué, ou le pointeur `NULL` en cas d'échec.

Donc si l'on souhaite allouer un espace de mémoire pour notre dit "tableau", on utiliserait `malloc` comme ceci :

Code : C

```c
pointeur = malloc(nombreElements * sizeof(*pointeur));
```

Ainsi on alloue une taille de : nombreElement x la taille d'un élément (représentée par `sizeof(*pointeur)`).



Attention :
Le nombre d'éléments doit représenter le nombre de cases de notre tableau, et donc doit être un **entier positif** (différent de zéro).

Règle de bon usage:
Il faut toujours tester le retour de malloc, calloc et realloc. Car en cas d'échec, elles retournent le pointeur `NULL`, qui entraîne des "segfaults" à son utilisation (notamment à l'utilisation de l'astérisque '*' sur le pointeur NULL).

La fonction calloc

Le prototype de la fonction calloc est :

Code : C

```c
void *calloc (size_t nmemb, size_t size);
```

- **nmemb** : le nombre d'éléments constituant l'espace mémoire à allouer.
- **size** : la taille en "byte" de chaque élément constituant l'espace mémoire à allouer.
- **La valeur retournée** : la fonction calloc retourne un pointeur sur le premier "byte" de l'espace alloué, ou le pointeur `NULL` en cas d'échec.

La différence avec la fonction malloc, est que calloc en plus de l'allocation, elle initialise l'espace alloué avec des 0 (elle met tous les bits à 0).

Il faut noter qu'elle est déconseillée pour allouer des espaces de type `float` ou `double` .

Pour allouer dynamiquement un tableau, on procèderait ainsi :

Code : C

```
pointeur = calloc(nombreElements , sizeof(*pointeur));
```

Ainsi la fonction calloc nous alloue un nombre d'éléments égal à 'nombreElements' du type pointé par 'pointeur'. Cet espace sera initialisé automatiquement par des zéros (0).

La fonction realloc

Elle permet de modifier la taille allouée pour un objet, son prototype est :

Code : C

```
void *realloc (void *ptr, size_t size);
```

- **ptr** : l'ancien espace mémoire alloué, et dont on voudrait modifier la taille. Si ce paramètre est `NULL` , alors la fonction se comporte comme malloc.
- **size** : la taille en "byte" du nouvel espace à allouer.
- **La valeur retournée** : la fonction realloc retourne un pointeur sur le premier "byte" du nouvel espace alloué, ou le pointeur `NULL` en cas d'échec.

Si l'on souhaite modifier la taille qu'on a allouée préalablement pour un objet, la fonction realloc s'utilise comme suit :

Code : C

```
pointeur = realloc(pointeur, nouvelleTaille);
```

Règles de bon usage:

Une utilisation telle qu'on vient de voir, est risquée. Pourquoi ?
Car en cas d'échec d'allocation, non seulement elle ne détruit pas l'ancien espace alloué, mais aussi elle retourne le pointeur NULL.

Dans l'expression pointeur = realloc(....), quelque soit le résultat, on aura écrasé l'ancienne valeur de pointeur. Donc on se retrouve dans un cas de mémoire déjà allouée, mais dont on ne connait plus où elle se trouve, ce qui vous rend incapable à la libérer; et c'est ce qu'on appel une fuite mémoire .

Solution :

Comme solution simple, on pourrait utiliser une variable intermédiaire, de type pointeur aussi :

Code : C

```
ptrintermediaire = realloc( pointeur , nouvelleTaille );

if(ptrintermediaire == NULL){
    free(pointeur);    //Desallocation
    //Ne pas oublier de notifier l'erreur.
}
```

```
        else{
            pointeur = ptrintermediaire ;
        }
```

La variable 'ptrintermediaire' doit être du même type que pointeur.
Ainsi nous n'écraserons l'ancienne valeur de pointeur que si la nouvelle allocation s'est bien passée.

Retourner un espace alloué dynamiquement

Si l'on souhaite effectuer notre allocation dynamique dans une fonction, pour la retourner à la fonction appelante. Il faudrait donc utiliser l'une des fonctions présentées ci-dessus (à savoir malloc, calloc ou realloc). Et de retourner le pointeur sur l'espace alloué.

Exemple :

Code : C

```c
int * fonctionAllocation(int nombreElements){
	int * ptr = malloc( nombreElements * sizeof(*ptr) );

	return ptr;
}

int main(void){
	int * ptr = fonctionAllocation(10);
	if(ptr == NULL)
		//........
	//.......
	return 0;
}
```

Il se peut que l'on souhaite allouer de la mémoire, sans utiliser le retour d'une fonction mais à l'aide d'un passage par référence (vous vous en rappelez ?).

Exemple :

Code : C

```c
void fonctionAllocation(int **ptr , int nombreElements){
	*ptr = malloc( nombreElements * sizeof(**ptr) );
}

int main(void){
	int * ptr;
	fonctionAllocation(&ptr , 10);
	if(ptr == NULL)
		//........
	//.......
	return 0;
}
```

Vous remarquerez que j'ai utilisé un double pointeur (int **), et oui c'est le piège . Avec l'utilisation d'un simple pointeur (int *), on serait entrain de faire un passage par valeur, et donc notre pointeur dans la fonction ne sera qu'une variable locale. Ainsi tous les changements qu'on apportera dessus seront de portée locale également (ne seront pas pris en compte dans la fonction main). D'où le double pointeur.

Les tableaux de tableaux ("tableaux à plusieurs dimensions")

Déclaration

Un tableau de tableaux (appelé tableau à plusieurs dimensions) se déclare par précision de la taille de chaque "dimension". Si par exemple je souhaite déclarer un tableau "tridimensionnel" alors je ferais ainsi :

Code : C

```
int t[taille1][taille2][taille3];
```

'taille1', 'taille2' et 'taille3' sont les tailles de chaque dimension. Autrement dit, 't' est un tableau de taille1 tableaux de taille2 tableaux de taille3 `int`s.

Utilisation

Initialisation

L'initialisation de ce type de tableaux n'est pas très différente de celle d'un tableau à une dimension :

Exemple 1 :

Code : C

```
int tableau1[2][3] = {{1,8,9},{0,6,4}};
```

Exemple 2 :

Code : C

```
int tableau1[2][3][2] = {{ {1,8}   , {0,6} , {0,0} },
                          { {31,52} , {4,8} , {11,5}}
                         };
```

Ou en ne précisant pas la première dimension :

Exemple :

Code : C

```
int tableau1[][3] = {{1,8,9},{0,6,4},{5,3,7},{2,2,2}};
```

Ainsi on a créé un tableau de 4x3 (équivalent à `int tableau[4][3]`), et initialisé ainsi :

1	8	9
0	6	4
5	3	7
2	2	2

Ou en n'initialisant que quelques cases du tableau :

Exemple :

Code : C

```
int tableau1[][3] = {{1,9},{0,4},{5,3,7},{2,2,2}};
```

Les cases restantes de chaque ligne seront donc initialisées à 0.

Ou pour initialiser toutes les cases à 0 :

Exemple :

Code : C

```
int tableau1[][3] = {{0},{0},{0},{0}};
```

Attention :

Code : C

```
int tableau1[][3] = {{0}};
```

Ceci est équivalent à `int tableau1[1][3]` et non pas `int tableau1[4][3]` !

Comme expliqué dans la partie "tableaux unidimensionnels", on pourrait également initialiser certaines cases de notre tableau; ceci s'applique bien évidemment à des tableaux multidimensionnels. Voici quelques exemples, je vous propose de les faire sous forme d'exercice pour voire si vous avez bien compris.

Exemple 1 :

Code : C

```
int t[4][5] = {{1,[3]=5,6},
              [2]={[3]1},
                 {2,4,[4]=10}};
```

Secret (cliquez pour afficher)

1	0	0	5	6
0	0	0	0	0
0	0	0	1	0
2	4	0	0	10

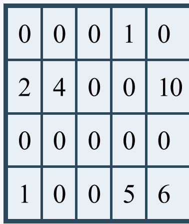
La première ligne a été initialisée suivant la règle que nous avons vu pour un tableau à une dimension (si vous ne vous en rappelez pas vous pouvez relire cette partie.
La deuxième ligne n'a pas été initialisée manuellement car nous avons sauté cette ligne pour aller directement à celle d'indice 2 ([2]={...}).
Naturellement l'initialisateur suivant est utilisé pour initialiser la ligne suivante donc celle d'indice 3.

Exemple 2 :

Code : C

```
int t[4][5] = {[3]={1,[3]=5,6},
               [0]={[3]=1},
                   {2,4,[4]=10}};
```

Secret (cliquez pour afficher)



0	0	0	1	0
2	4	0	0	10
0	0	0	0	0
1	0	0	5	6

Exemple 3 :

Code : C

```
int t[][5] = {  [1]={1,[3]=5,6},
                [2]={7,9,[1]=1,5},
                    {2,4,[4]=10},
                    {0,7,5,3,8,4,7},
                    {0}};
```

Secret (cliquez pour afficher)

Il se peut que vous ayez un warning vous indiquant que vous avez dépassé la taille pour l'initialisateur {0,7,5,3,8,4,7}.

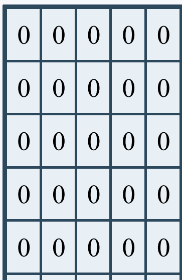
0	0	0	0	0
1	0	0	5	6
7	1	5	0	0
2	4	0	0	10
0	7	5	3	8
0	0	0	0	0

Exemple 4 :

Code : C

```
int t[][5] = {[5]={0}};
```

Secret (cliquez pour afficher)



0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

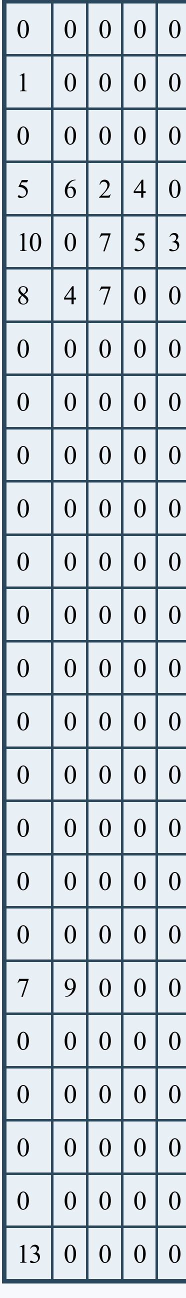
0	0	0	0	0

Exemple 5 :

Code : C

```c
int t[][5] =
{[1]=1,[3]=5,6,2,4,[4]=10,0,7,5,3,8,4,7,[18]=7,9,[23]=13};
```

Secret (cliquez pour afficher)



0	0	0	0	0
1	0	0	0	0
0	0	0	0	0
5	6	2	4	0
10	0	7	5	3
8	4	7	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
7	9	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
13	0	0	0	0

Parcours

Le parcours d'un tableau à plusieurs dimensions s'effectue en précisant l'indice pour chaque dimension.

Pour un exemple de tableau de trois dimensions, et à l'aide de trois boucles imbriquées, on arriverait à accéder à toutes les cases

de notre tableau.

Exemple :

Code : C

```
int tableau[taille1][taille2][taille3];
int i,j,k;

for(i=0 ; i < taille1 ; i++){                  //Première dimension
        for(j=0 ; j < taille2 ; j++){          //Deuxième dimension
                for(k=0 ; k < taille3 ; k++){  //troisième dimension
                        tableau[i][j][k] = 0;
                }
        }
}
```

Les pointeurs sur tableaux

Vous pensez peut être au double pointeur (int **) , non ce n'est pas de cela qu'il s'agit.

La déclaration d'un pointeur sur tableau s'effectue en utilisant des parenthèses, l'astérisque '*' et en définissant la taille du tableau sur lequel on souhaite pointer. Notez que les parenthèses sont très importantes car en leur absence, ce sera un tableau de pointeurs qu'on aura déclaré, ce qui n'est pas la même chose.

Exemple :

Code : C

```
int (*ptrtableau)[4];
```

Dans cet exemple, il s'agit d'une déclaration d'un pointeur sur tableaux de 4 entiers.

L'erreur à ne pas faire :

Code : C

```
int * ptrtableau[4];
```

Ici il ne s'agit pas d'un pointeur sur tableau de 4, mais un tableau de 4 pointeurs sur int (dont l'explication est dans la partie suivante).

Utilité des pointeurs sur tableaux

C'est un moyen très pratique pour déclarer un tableau à deux dimensions . Sachant qu'une dimension est déjà pré allouée, il ne reste plus qu'à allouer la deuxième dimension.

Exemple :

Code : C

```
int (*tableau)[4];

tableau = malloc(5 * sizeof(*tableau));
```

Ainsi on aura déclaré un tableau de 5 tableaux de 4 entiers chacun (équivalent à `int tableau[5][4];`).

Libération de mémoire allouée

Pour libérer la mémoire, on doit utiliser la fonction free toujours, et lui donner en paramètre le pointeur sur l'espace alloué à l'aide de malloc :

Code : C

```
int (*tableau)[4];

//----Allocation-----
tableau = malloc(5 * sizeof(*tableau));
if(tableau == NULL){
    //Notifier l'erreur
    exit(EXIT_FAILURE);
}

//----Libération en cas d'allocation réussie-----
free(tableau);
```

La deuxième dimension sera libérée automatiquement donc pas besoin de free pour le faire.
Ainsi on remarque que ce type est beaucoup plus rapide d'utilisation (en terme d'allocation/libération de mémoire) du fait qu'il ne nécessite pas de boucles.

Les tableaux de pointeurs

Un tableau de pointeurs, est un outil pour ranger un ensemble de pointeurs sur différentes variables.
Il peut également servir pour déclarer un tableau à deux dimensions, en allouant plusieurs espaces et stockant leurs pointeurs dans notre tableau.

Exemple de déclaration :

Code : C

```
int * tableauDePtr[5];
```

Exemple d'utilisation pour déclarer un tableau à deux dimensions :

Code : C

```
int * tableauDePtr[5];
int i;

for(i=0 ; i < 5 ; i++){
        tableauDePtr[i] = malloc(4 * sizeof(tableau[0]));
}
```

Ainsi on aura créé un tableau de 5 tableaux de 4 entiers chacun (équivalent à `int tableau[5][4];`).

Libération de mémoire allouée

La libération de mémoire allouée doit se faire en parcourant le tableau, et en allouant pointeur par pointeur dans ce tableau :

Code : C

```
int i;
int * tableauDePtr[5];

//---Allocation-----
for(i=0 ; i < 5 ; i++){
        tableauDePtr[i] = malloc(4 * sizeof(tableau[0]));
        if(tableauDePtr[i] == NULL){     //En cas d'erreur
d'allocation
                //N'oubliez pas de notifier l'erreur
                for(i=i-1 ; i >= 0 ; i--)   //Libération de l'espace
déjà alloué
                        free(tableauDePtr[i]);

                exit(EXIT_FAILURE);
        }
}

//---Libération en cas d'allocation réussie-----
for(i=0 ; i < 5 ; i++){
        free(tableauDePtr[i]);
}
```

L'erreur à ne pas faire :

Code : C

```
free(tableauDePtr);
```

tableauDePtr est un type automatique, donc sera libéré automatiquement à la fin de la fonction dans laquelle il est déclaré. Faites donc attention à ne pas le confondre avec le type pointeur sur tableaux qu'on a vu précédemment.

Allocation dynamique

Cette partie je la qualifie étant la plus dure à suivre, donc mettez vos ceintures .

L'allocation dynamique d'un dit "tableau" à plusieurs dimensions, s'effectue en allouant les dimensions une par une. Si je prends l'exemple d'un tableau tridimensionnel, il faudrait un triple pointeur pour y arriver. Donc veillez à avoir le même nombre d'astérisques '*' dans la déclaration de pointeur, que le nombre de dimensions de votre tableau.

Si je prends un exemple de tableau à 3 dimensions que je souhaite allouer dynamiquement, j'utiliserais un pointeur déclaré ainsi :

Code : C

```
int ***ptr;     //3 étoiles pour 3 dimensions
```

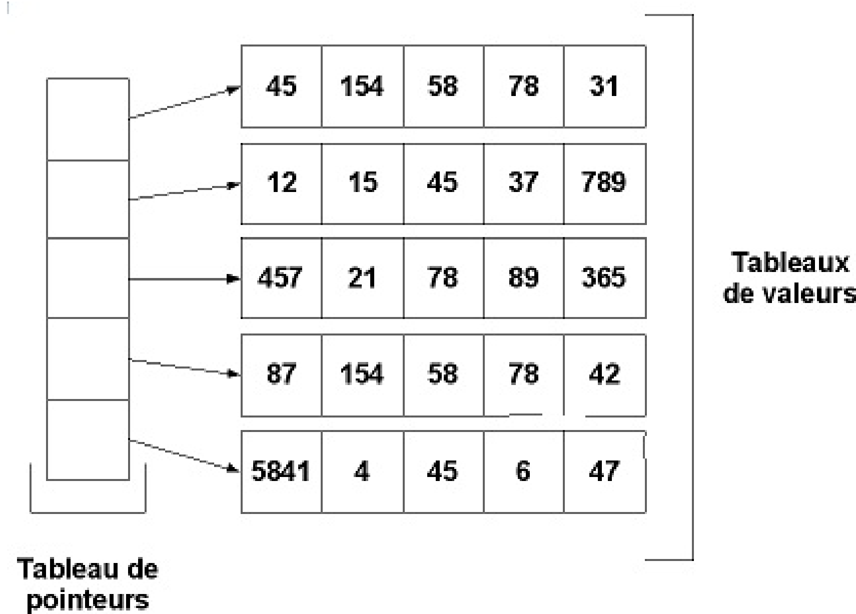
Et un tableau à deux dimensions nécessiterait un double pointeur :

Code : C

```
int **ptr;     //2 étoiles pour 2 dimensions
```

Tableaux bidimensionnels

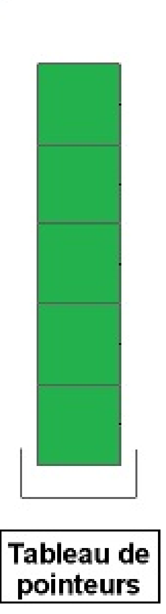
Ce type de tableaux, est un simple tableau de pointeurs, chacun de ces pointeurs va pointer sur un espace représentant un tableau à une dimension, une petite image est la bienvenue je pense :



L'allocation doit se faire en respectant la démarche suivante :

- 1- Allocation de la première dimension (ptr) :

  Il s'agit du tableau de pointeurs qui contiendra plusieurs pointeurs qui à leur tour pointent sur des espaces alloués sous forme de tableaux.

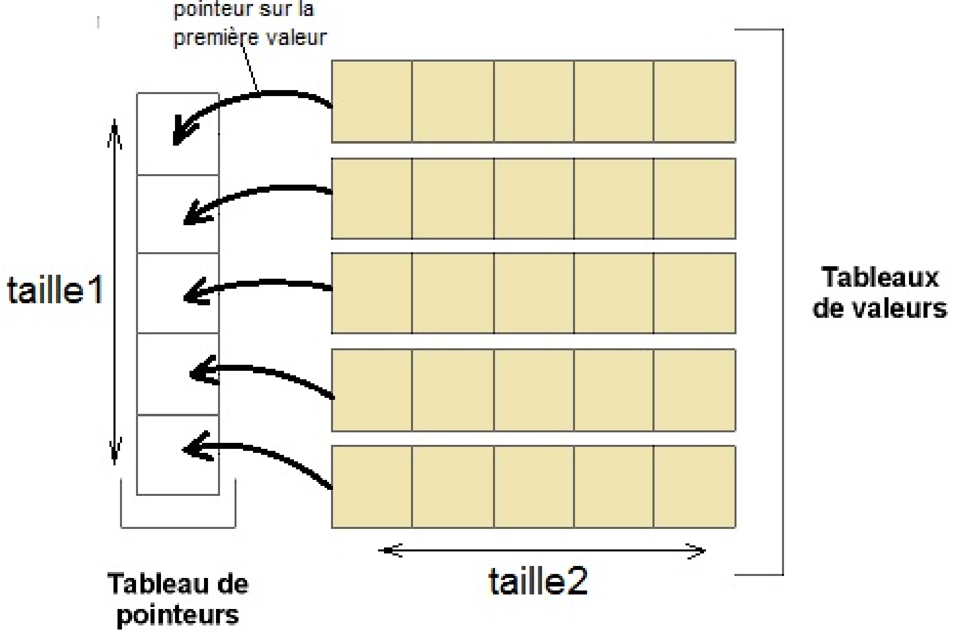


Code : C

```
int **ptr;
```

```
	ptr = malloc(taille1 * sizeof(*ptr));		//On alloue 
'taille1' pointeurs.
	if(ptr == NULL)
		//Ne pas oublier de notifier l'erreur et de quitter le 
programme.
```

- 2- Allocation de la deuxième dimension (*ptr) :



Il s'agit des différents tableaux appelés "tableaux de valeurs" sur l'image ci-dessus.

Code : C

```
	int i;

	for(i=0 ; i < taille1 ; i++){
		ptr[i] = malloc(taille2 * sizeof(*(ptr[i])));		//On 
alloue des tableaux de 'taille2' variables.
		if(ptr[i] == NULL){
			//Il faut libérer la mémoire déjà allouée
			//Ne pas oublier de notifier l'erreur et de quitter le 
programme.
		}
	}
```

Ainsi on obtient le code suivant :

Code : C

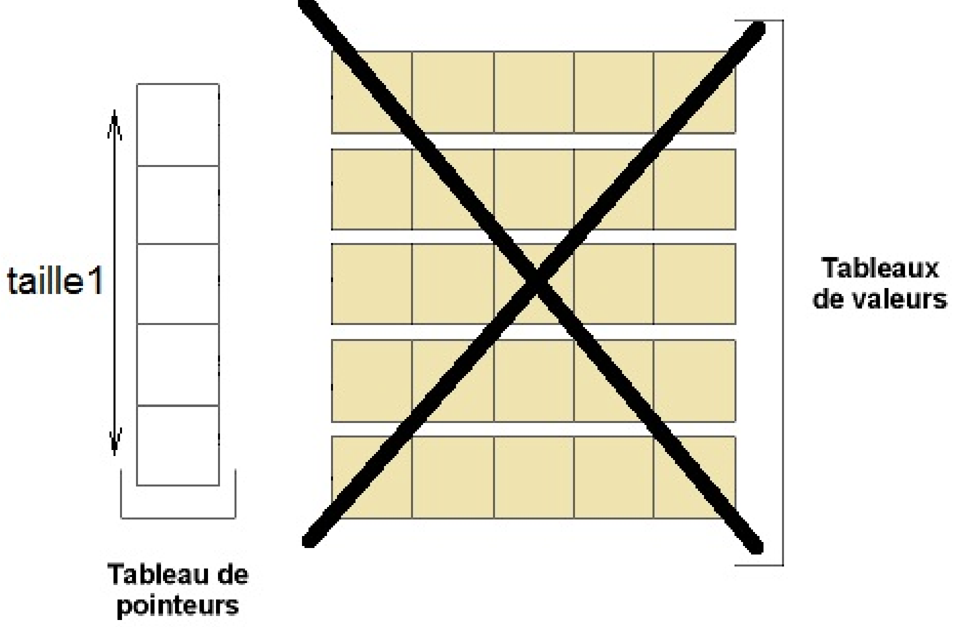
```
	int i , taille1 = 2 , taille2 = 3;
	int **ptr;

	ptr = malloc(taille1 * sizeof(*ptr));		//On alloue 'taille1' 
pointeurs.
```

```
if(ptr == NULL)
      //Ne pas oublier de notifier l'erreur et de quitter le programme.

for(i=0 ; i < taille1 ; i++){
     ptr[i] = malloc(taille2 * sizeof(**ptr) );        //On alloue des tableaux de 'taille2' variables.
     if(ptr[i] == NULL){                               //En cas d'erreur d'allocation
         //Il faut libérer la mémoire déjà allouée
         //Ne pas oublier de notifier l'erreur et de quitter le programme.
     }
}

//notre tableau ptr[2][3] est maintenant utilisable...
```

Quant à la libération de mémoire elle se fait suivant l'ordre inverse à l'allocation :

- 1- Libération de la deuxième dimension (*ptr) :

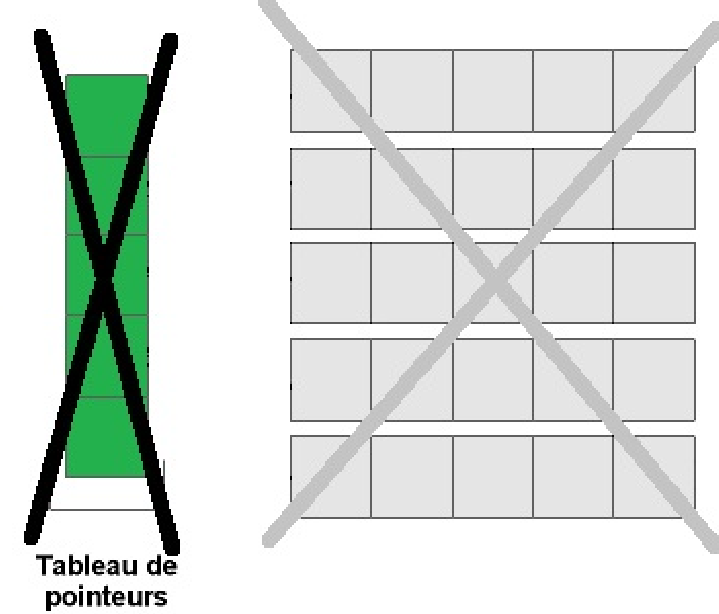
  En utilisant une boucle, on doit donc libérer les différents tableaux à une dimension qu'on a alloués :



Code : C

```
int i;

for(i=0 ; i < taille1 ; i++){
     free(ptr[i]);
}
```

- 2- Libération de la première dimension (ptr) :



Code : C

```
free(ptr);
```

On obtient donc le code suivant :

Code : C

```
int i;

//Après allocation....

//--------La libération---------
for(i=0 ; i < taille1 ; i++){
    free(ptr[i]);
}

free(ptr);
ptr = NULL;   //Ceci est par mesure de sécurité (ce n'est donc pas
obligatoire).
//---------------------------------
```

Cette libération n'est utilisable que si l'allocation s'est bien déroulé (sans erreurs), Que faire donc en cas d'erreur d'allocation ?

Et bien il s'agit des `if( ptr[i] == NULL)` que vous avez peut être remarquées dans le code d'allocation . On va maintenant essayer de les remplir avec ce qu'il faut.

- 1- Première dimension :

Code : C

```
int **ptr;

ptr = malloc(taille1 * sizeof(*ptr));        //On alloue
'taille1' pointeurs.
if(ptr == NULL)
      //Pas de libération nécessaire à ce niveau, il suffirait
donc de notifier l'erreur et de fermer le programme (par
utilisation de exit(-1) par exemple).
```

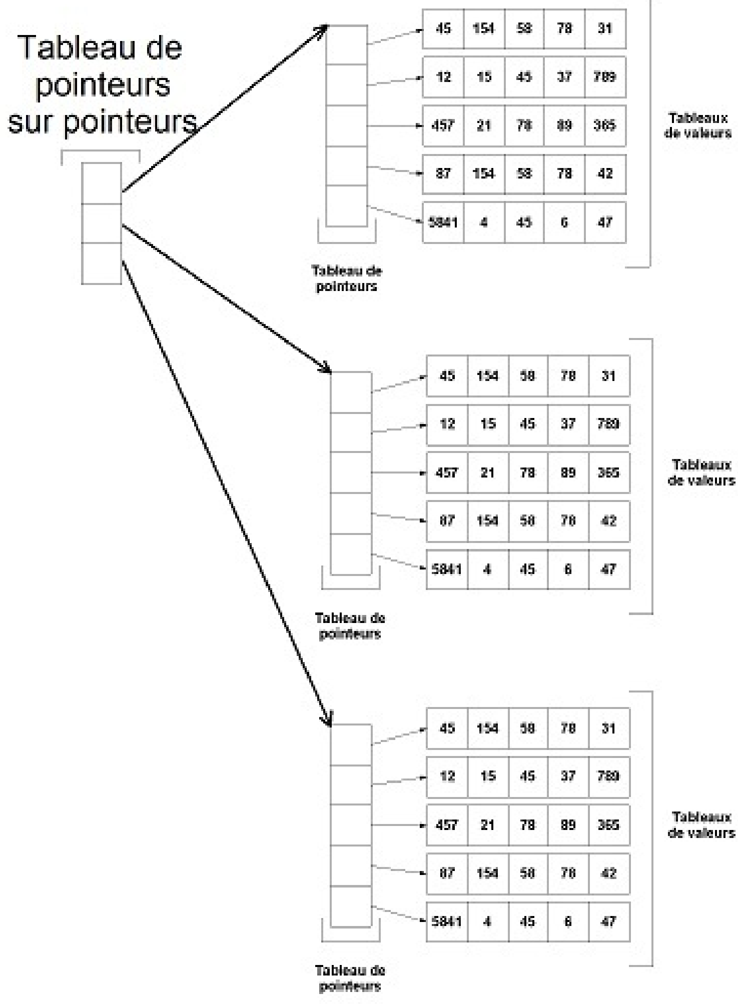
- 2- Deuxième dimension :

Code : C

```
int i;

for(i=0 ; i < taille1 ; i++){
     ptr[i] = malloc(taille2 * sizeof(*(ptr[i])));        //On
alloue des tableaux de 'taille2' variables.
     if(ptr[i] == NULL){
         for(i = i-1 ; i >= 0 ; i--)     //On parcourt la boucle
dans l'ordre inverse pour libérer ce qui a déjà été alloué
              free(ptr[i]);
         free(ptr);                      //On libère la première
dimension.

         //Ne pas oublier de notifier l'erreur et de quitter le
programme (en utilisant exit(-2) par exemple).
     }
}
```

Tableaux tridimensionnels

Passant maintenant à un tableau à 3 dimensions . Il s'agit d'un tableau de pointeurs sur pointeurs, chacun d'eux va pointer sur un pointeur qui lui pointe sur un tableau bref, trop de pointeurs et de blabla, une image serait donc plus explicite :



L'allocation dynamique de ce tableau, sera divisée en 3 étapes qui doivent correspondre à l'ordre suivant :

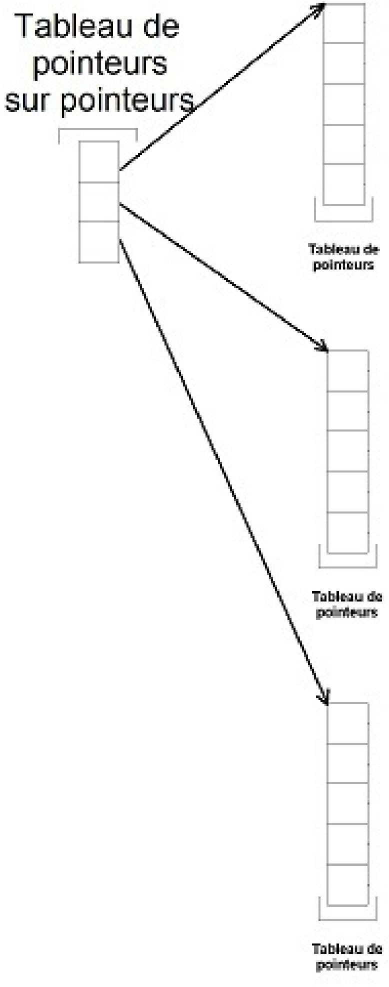
- 1- Allocation de la première dimension, représentée par ptr : Par analogie aux tableaux automatiques, ce sera celle qui se trouve entre crochets ([]) à gauche.

Code : C

```c
int ***ptr;
ptr = malloc(taille1 * sizeof(*ptr));
```

- 2- allocation de la deuxième dimension, représentée par *ptr : Par analogie aux tableaux automatiques, ce sera celle qui se

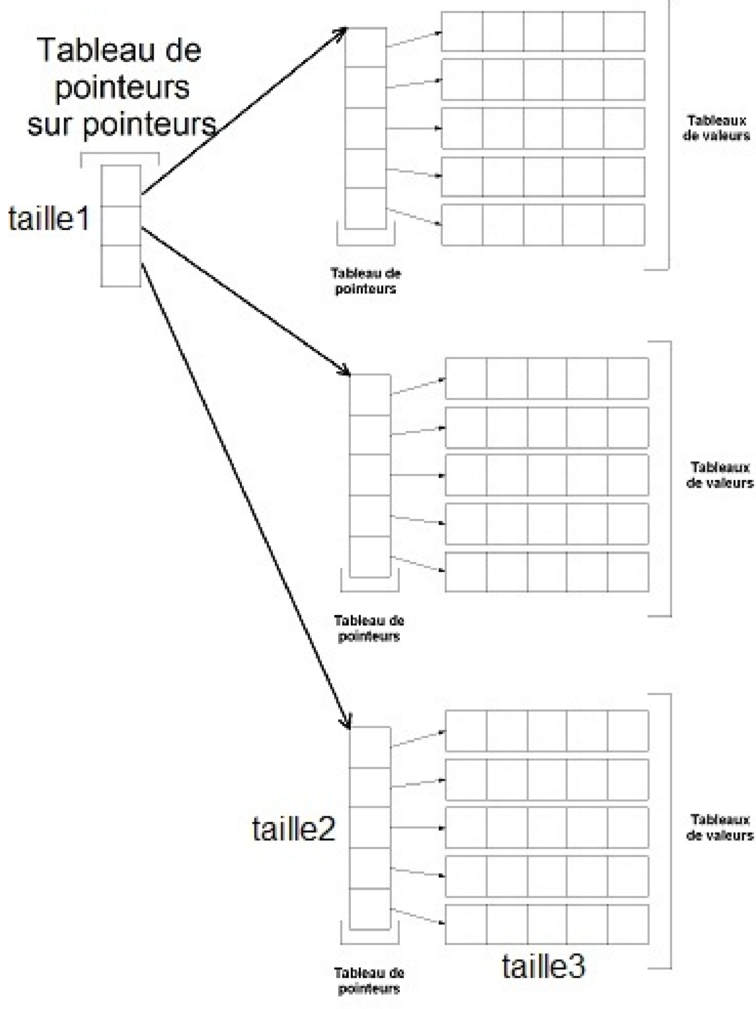
trouve entre crochets ([]) au milieu.



Code : C

```c
for(i=0 ; i < taille1 ; i++){
     ptr[i] = malloc(taille2 * sizeof(**ptr));
}
```

- 3- allocation de la troisième dimension, représentée par **ptr : Par analogie aux tableaux automatiques, ce sera celle qui se trouve entre crochets ([]) à droite.



Code : C

```
for(i=0 ; i < taille1 ; i++){
     for(j=0 ; j < taille2 ; j++){
          ptr[i][j] = malloc(taille3 * sizeof(***ptr));
     }
}
```

Nous nous retrouvons donc avec le code suivant :

Code : C

```
int ***ptr;
int i,j;

ptr = malloc(taille1 * sizeof(*ptr));
if(ptr == NULL)          //Pas de libération de mémoire à ce niveau
     return -1;          //Exemple de code d'erreur

for(i=0 ; i < taille1 ; i++){
     ptr[i] = malloc(taille2 * sizeof(**ptr));
     if( ptr[i] == NULL) //Pensez à libérer la mémoire déjà allouée
et fermer le programme.
          return -1;     //Exemple de code d'erreur
```

```
        }
        for(i=0 ; i < taille1 ; i++){
             for(j=0 ; j < taille2 ; j++){
                     ptr[i][j] = malloc(taille3 * sizeof(***ptr));
                     if(ptr[i][j] == NULL)    //Pensez à libérer correctement
        la mémoire déjà allouée
                            return -1;            //Exemple de code d'erreur
             }
        }
```

Pour un tableau de dimensions supérieurs, il faut procéder de la même manière, en allouant les dimensions une à une, et en respectant l'ordre des allocations.

Pour la libération de mémoire déjà allouée, il faut que cela soit fait dans l'ordre inverse à celui de l'allocation. Pour le même exemple présenté ci-dessus on procèderait ainsi :

- 1- Libération de l'espace **ptr.

  Code : C

  ```
  for(i=0 ; i < tailleDejaAllouee1 ; i++){
       for(j=0 ; j < tailleDejaAllouee2 ; j++){
              free(ptr[i][j]);
       }
  }
  ```

- 2- Libération de l'espace *ptr.

  Code : C

  ```
  for(i=0 ; i < tailleDejaAllouee1 ; i++){
       free(ptr[i]);
  }
  ```

- 3- Libération de l'espace ptr.

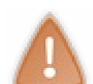
  Code : C

  ```
  free(ptr);
  ```

Le code de libération est donc comme ceci :

Code : C

```
for(i=0 ; i < tailleDejaAllouee1 ; i++){
     for(j=0 ; j < tailleDejaAllouee2 ; j++){
            free(ptr[i][j]);
     }
     free(ptr[i]);
}
free(ptr);
```



Attention :
Cette méthode de libération n'est valable que si l'allocation s'est **bien déroulée**.

Voici un code montrant la méthode de libération de mémoire si l'allocation échoue avant d'allouer le tableau entièrement :

Secret (cliquez pour afficher)

Code : C

```
#include <stdlib.h>

void * my_free(int***,int,int);

void * my_free(int ***ptr,int tailleDejaAllouee1,int
tailleDejaAllouee2){
    int i,j;

    for(i=0 ; i < tailleDejaAllouee1 ; i++){
         for(j=0 ; j < tailleDejaAllouee2 ; j++){
               free(ptr[i][j]);
         }
         free(ptr[i]);
    }
    free(ptr);

    return NULL;
}

int main(void)
{
    /*---------------------------------------------------*/
    int taille1 = 2, taille2 = 2, taille3 = 2;
    int ***ptr;
    int i,j;

    /*-----------------Allocation------------------------*/
    ptr = malloc(taille1 * sizeof(*ptr));
    if(ptr == NULL)
         return -1;

    for(i=0 ; i < taille1 ; i++){
         ptr[i] = malloc(taille2 * sizeof(**ptr));

         if( ptr[i] == NULL){         //Erreur d'allocation

              for(--i ; i>=0 ; i--)   //On libère l'espace déjà
alloué
                   free(ptr[i]);

              free(ptr);

              return -1;              //Fin du programme
         }
    }

    for(i=0 ; i < taille1 ; i++){
         for(j=0 ; j < taille2 ; j++){
               ptr[i][j] = malloc(taille3 * sizeof(***ptr));

               if(ptr[i][j] == NULL){          //Erreur
d'allocation

                    for(--j ; j >= 0 ; j--)     //On libère
l'espace déjà alloué
```

```
                        free(ptr[i][j]);

                    free(ptr[i]);

                    my_free(ptr , i , taille2);

                    for(--i ; i >= 0 ; i--)
                        free(ptr[i]);

                    return -2;                //Fin du programme
                }
            }
        }

    /*---------------Libération-------------------------*/
    ptr = my_free(ptr,taille1,taille2);       //On libère la
mémoire et on met la valeur de ptr à NULL
    /*--------------------------------------------------*/

    return 0;
}
```

Attention :

On pourrait optimiser nos boucles d'allocation comme ceci :

Secret (cliquez pour afficher)

Code : C

```
int ***ptr;
int i,j;

ptr = malloc(taille1 * sizeof(*ptr));
if(ptr == NULL)          //Pas de libération de mémoire à
ce niveau
    return -1;           //Exemple de code d'erreur

for(i=0 ; i < taille1 ; i++){
    ptr[i] = malloc(taille2 * sizeof(**ptr));
    if( ptr[i] == NULL) //Pensez à libérer la mémoire déjà
allouée et fermer le programme.
        return -1;      //Exemple de code d'erreur

    for(j=0 ; j < taille2 ; j++){
        ptr[i][j] = malloc(taille3 * sizeof(***ptr));
        if(ptr[i][j] == NULL)    //Pensez à libérer
correctement la mémoire déjà allouée
            return -1;           //Exemple de code
d'erreur
    }
}
```

Cependant le problème se posera dans la libération de mémoire déjà allouée qui sera beaucoup plus compliquée :-) . Si vous n'êtes pas convaincu de la complexité que cela peut engendrer, essayer de le faire, et faîtes vous corrigé par une personne confirmée.

La méthode que je viens d'expliquer pour un tableau à trois dimensions, est valable pour un tableau à plus de dimensions, il suffirait de suivre la même logique à savoir allouer dimension par dimension en respectant l'ordre. Une

fois à l'aise avec les allocations, vous pouvez bien entendu optimiser vos boucles d'allocations .

Fonctions : mettre un tableau en argument

Si vous pensez à utiliser le double pointeur (int **), non il ne s'agit pas de cela.

L'erreur à ne pas faire :

Code : C

```c
void ma_fonction(int ** tableau2D){
	//....
	tableau2D[i][j] = 10;
}

int main(void){
	int tableau2D[10][5];

	ma_fonction(tableau2D);
	return 0;
}
```

Si le tableau est alloué automatiquement, il ne faut pas l'envoyer à une fonction attendant un pointeur sur pointeur (cf : code ci-dessus).
Les méthodes correctes correspondent globalement aux différentes déclarations d'un tableau telles que nous l'avons vu plus haut.

Méthode classique

Lors de la déclaration des arguments d'une fonction, le type tableau à plusieurs dimensions est désigné par des crochets [], avec ou sans taille.

Code : C

```c
void ma_fonction(int tableau[taille1][taille2][taille3]){
	//Corps de la fonction...
}
```

Important :

Une remarque importante à noter est qu'il faut **au plus** une dimension avec une taille non spécifiée.
Une déclaration de tableau en argument d'une fonction comme ceci :

Code : C

```c
void ma_fonction(int tableau[][][taille3]){
	//Corps de la fonction...
}
```

est incorrecte !

Exemple de déclaration correcte :

Code : C

```
void ma_fonction(int tableau[][taille2][taille3]){
      //Corps de la fonction...
}
```

Et l'appel sera par simple envoi du nom du tableau, c'est valable pour les deux cas, à savoir avec ou sans précision de la taille de la première dimension :

Code : C

```
int tableau[taille1][taille2][taille3];  //La déclaration du tableau

ma_fonction(tableau);   //L'appel à la fonction avec passage d'un tableau en argument.
```

Utiliser un pointeur sur tableau

Exemple :

Code : C

```
int fonction(int (*matrice)[3]){
        //......
        matrice[0][2] = 15;
}
```

Dans cet exemple matrice est un pointeur sur tableau de 3 `int`.
Il permettra de réceptionner la matrice envoyée sous forme de pointeur sur le premier élément (en l'occurrence, pointeur sur tableau de trois entiers) lors de la l'appel à cette fonction.
Exemple :

Secret (cliquez pour afficher)

Code : C

```
int fonction(int (*matrice)[3]){
        //......
        matrice[0][2] = 15;
}
int main(void){
        int matrice[4][3];  //Notez que la taille 3 correspond à celle spécifiée dans la déclaration de la fonction!

        fonction(matrice);
        return 0;
}
```

Jusqu'ici, vous avez peut être remarqué que si l'on souhaite parcourir la matrice passée en argument, on serait obligé d'avoir une information sur la taille de la matrice. Vous pensez peut être à utiliser l'opérateur `sizeof` ? Alors je vous mets de suite en garde, cela ne fonctionnera pas (du moins pour la première dimension).

Code : C

```
void fonction(int tab[][3]){
        sizeof tab;
}
```

Qui est équivalent à :

Code : C

```
void fonction(int tab[5][3]){
        sizeof tab;
}
```

Qui est équivalent à :

Code : C

```
void fonction(int (*tab)[3]){
        sizeof tab;
}
```

Contrairement à ce qu'on pourrait s'attendre à voir, dans ces trois cas le résultat de l'opérateur `sizeof` ne donnera pas la taille des tableaux déclarés en paramètre, mais celle d'un pointeur (souvent 4 octets). Ce qui démontre (en quelque sorte) que le passage d'un paramètre formel de type 'tableau' n'est pas faisable à partir des déclarations ci-dessus.
Dans le cas général, on préfère passer la taille de notre matrice (ou au moins la taille des dites "lignes") en paramètre à la fonction.

Secret (cliquez pour afficher)

Code : C

```
void fonction1(int matrice[][3], size_t Nlignes){
        sizeof matrice[0] * Nlignes;   //Ceci permet de récupérer la taille de toute la matrice.
        sizeof *matrice * Nlignes;     //équivalente à celle ci-dessus
}
void fonction2(int matrice[10][3], size_t Nlignes){ //La taille 10 ici sera ignorée, donc est inutile. Contrairement à la taille 3 qu'il faudra respecter!
        sizeof matrice[0] * Nlignes;   //Ceci permet de récupérer la taille de toute la matrice.
        sizeof *matrice * Nlignes;     //équivalente à celle ci-dessus
}
void fonction3(int (*matrice)[3], size_t Nlignes){
        sizeof matrice[0] * Nlignes;   //Ceci permet de récupérer la taille de toute la matrice.
        sizeof *matrice * Nlignes;     //équivalente à celle ci-dessus
}
int main (void){
        int matrice[4][3];
        fonction1(matrice,4);
        fonction2(matrice,4);
        fonction3(matrice,4);
}
```

Ainsi nous disposerons de toutes les informations nécessaires pour parcourir notre matrice à l'aide d'une boucle.

Utilisation d'un double pointeur `int **ptr`

Oui je vous ai mentit en vous disant qu'il n'est pas possible de réceptionner une matrice statique à partir d'un double pointeur. Mais cela n'est tout de même pas si simple à manier, et demanderait un peu de bricolage pour y parvenir, vous pouvez en juger vous même.

La méthode consisterait à réceptionner l'adresse envoyée lors de l'appel de la fonction, qui représente l'adresse mémoire du premier élément de la première dimension de la matrice. Et à l'aide d'un tableau intermédiaire de pointeurs, ainsi que de la mise en

équation des adresses mémoire de toutes les cases de la matrice, on arriverait à manier correctement une matrice 2D.
Je vous propose d'analyser ce code :

Code : C

```
int fonction(int **mat)
{
	int i, j, *index[3];   //Index est le tableau de pointeurs qu'on utilisera sous forme de mémoire tampon.

	for (i = 0 ; i < 3 ; i++)  //On initialise notre tableau intermédiaire 'index'
		index[i] = (int *)mat + 3 * i;

	for(i = 0 ; i < 3 ; i++)
	{
		printf("\n");
		for(j = 0 ; j < 3 ; j++)
		{
		 printf("%d", index[i][j]);
		}
	}
	printf("\n");

	return 0;
}
```

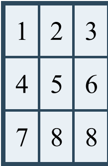
Pour bien comprendre le pourquoi du comment, je vous fais un petit rappel (qui ne sera pas qu'un simple rappel pour certains).

Imaginons que l'on veuille mettre en équation les adresses des cases constituant une matrice, afin de pouvoir les indexer en utilisant uniquement comme données l'adresse du premier élément dans cette matrice ainsi que sa taille (lignes et colonnes).
Avant d'attaquer les explications, je tiens à vous informer/rappeler que le langage C garantie qu'un tableau à deux dimensions sera placé en mémoire selon la disposition Row-major order.

En quoi cela nous intéresse ?

Cela signifie que seulement à partir de l'adresse du premier élément, et en connaissant la taille d'une matrice, on arriverait à connaître l'adresse de chacune de ses cases.

Exemple :



1	2	3
4	5	6
7	8	8

Cette matrice est ramenée dans la mémoire à la forme :

1	2	3	4	5	6	7	8	8

(disposition Row-major order).

Pour la suite de l'explication, nous allons prendre trois variables de type pointeurs avec les noms adresse_1, adresse_2 et adresse_3, qui sont respectivement les adresses en mémoire des nombres 1, 4 et 7.
Nous allons dans un premier temps récupérer l'adresse de la première case (celle contenant le chiffre 1), puis nous allons nous décaler de trois cases (trois étant le nombre de colonnes de la matrice) pour obtenir les adresses des élément 4 et 7;

Code : C

```c
int matrice[3][3] = {{1,2,3},{4,5,6},{7,8,9}}; //La matrice
int **adresse_matrice = (int**)matrice;  //Ceci est pour simuler un
passage d'une matrice à une fonction
int *adresse_1, *adresse_2, *adresse_3;

adresse_1 = (int*)adresse_matrice;
adresse_2 = (int*)adresse_matrice + 3;
adresse_3 = (int*)adresse_matrice + 6;
```

Ensuite nous allons regrouper maintenant nos pointeurs adresse_1/2/3 dans un tableau :

Code : C

```c
int matrice[3][3] = {{1,2,3},{4,5,6},{7,8,9}}; //La matrice
int **adresse_matrice = (int**)matrice;  //Ceci est pour simuler un
passage d'une matrice à une fonction
int *adresse[3];                      //Les pointeurs intermédiaires

adresse[0] = (int*)adresse_matrice + (0*3);
adresse[1] = (int*)adresse_matrice + (1*3);
adresse[2] = (int*)adresse_matrice + (2*3);
```

D'où le code de la fonction présenté précédemment.
Secret (cliquez pour afficher)

Code : C

```c
int fonction(int **mat)
{
        int i, j, *index[3];   //Index est le tableau de
pointeurs qu'on utilisera sous forme de mémoire tampon.

        for (i = 0 ; i < 3 ; i++)  //On initialise notre tableau
intermédiaire 'index'
                index[i] = (int *)mat + 3 * i;

        for(i = 0 ; i < 3 ; i++)
        {
                printf("\n");
                for(j = 0 ; j < 3 ; j++)
                {
                 printf("%d", index[i][j]);
                }
        }
        printf("\n");

        return 0;
}
```

En conclusion

Cette méthode n'est pas utilisée dans la pratique, de part son caractère complexe et bricolé, sans oublier qu'elle fait appel à un tableau intermédiaire, et peut poser des problèmes du fait qu'on fait des affectations aux pointeurs avec types différents.
Je ne vous conseillerai donc pas de l'utiliser dans le cas général; je tenais simplement à vous la présenter à titre informatif, ça nous a aussi permis de voir la nature d'une tableau multidimensionnel en mémoire.

Utilisation d'un typedef

L'utilisation du typedef n'est qu'un raccourci, pour alléger les déclarations de tableaux.

Exemple :

Code : C

```
typedef int tableau3x3x3[3][3][3];

void ma_fonction1(tableau3x3x3 tab){    //Déclaration d'un argument
de type tableau tridimensionnel de 3x3x3
      //Corps de la fonction...
      printf("%zu\n",sizeof(tab[0][0]));//Pour un test
}

int main(void)
{
    tableau3x3x3 tableau;     //Déclaration du tableau 3D

    ma_fonction1(tableau);    //Appel de la fonction en lui passant
notre tableau en paramètre
    return 0;
}
```

Fonctions : retourner un tableau

Comme nous l'avons vu dans la partie des tableaux unidimensionnels, il est interdit de renvoyer un tableau automatique (alloué statiquement) :

Code : C

```
int (ma_fonction(void))[2][3][4]{
     int tableau[2][3][4];

     return tableau;
}
```

Il est donc impératif de procéder à une allocation dynamique, puis de retourner le pointeur sur l'espace alloué dynamiquement (tel que nous l'avons vu avant dans la partie d'allocation dynamique).

Exemple :

Secret (cliquez pour afficher)

Code : C

```
int *** fonction_allocation(int taille1, int taille2, int
taille3){
     int ***ptr;
     int i,j;

     ptr = malloc(taille1 * sizeof(*ptr));
     if(ptr == NULL)            //Pas de libération de mémoire à ce
niveau
          return -1;            //Exemple de code d'erreur

     for(i=0 ; i < taille1 ; i++){
          ptr[i] = malloc(taille2 * sizeof(**ptr));
          if( ptr[i] == NULL) //Pensez à libérer la mémoire déjà
allouée et fermer le programme.
               return -1;     //Exemple de code d'erreur
     }
```

```
	for(i=0 ; i < taille1 ; i++){
		for(j=0 ; j < taille2 ; j++){
			ptr[i][j] = malloc(taille3 * sizeof(***ptr));
			if(ptr[i][j] == NULL)	//Pensez à libérer
correctement la mémoire déjà allouée
				return -1;	//Exemple de code
d'erreur
		}
	}
	return ptr;				//retour du pointeur sur
l'espace alloué
}

int main(void){
	int taille1 = 2, taille2 = 2, taille3 = 2;
	int *** ptr = fonction_allocation(taille1,taille2,taille3);
	//.....
	ptr = my_free(ptr,taille1,taille2);	//Cette fonction est
décrite dans la partie allocation dynamique
	return 0;
}
```

Exercices

Pointeurs

Exercice 1

- 1. Déclarer un pointeur sur int et l'initialiser par le pointeur NULL.
- 2. Déclarer un pointeur sur pointeur sur int.
- 3. Déclarer une variable de type int.
- 4. Initialiser le pointeur sur int par l'adresse de cette variable.
- 5. Initialiser le pointeur sur pointeur sur int par l'adresse du pointeur sur int.
- 6. Allouer dynamiquement un espace de mémoire suffisant pour contenir une variable de type int, et stocker son adresse dans le pointeur sur int.

Solution

Secret (cliquez pour afficher)

Code : C

```
//Q1 :
int * pointeur = NULL;
//Q2 :
int ** pointeurSurPointeur;
//Q3 :
int variable;
//Q4 :
pointeur = &variable;
//Q5 :
pointeurSurPointeur = &pointeur;
//Q6 :
pointeur = malloc(sizeof(int));
//Ou
pointeur = malloc(sizeof( *pointeur ));
```

Exercice 2

- 1. Déclarer deux pointeurs sur `float` .
- 2. Déclarer deux variables de type `float` .
- 3. Initialiser chacun des pointeurs avec les adresses des deux variables `float` .
- 4. Affecter la valeur 12.5 à la première variable en utilisant son pointeur.
- 5. Affecter la valeur 5.76 à la deuxième variable en utilisant son pointeur.
- 6. Échanger le contenu des deux variables en utilisant leurs pointeurs.

Secret **(cliquez pour afficher)**

Code : C

```
//Q1 :
float * ptr1 , * ptr2 ;
//Q2 :
float variable1 , variable2 ;
//Q3 :
ptr1 = &variable1;
ptr2 = &variable2;
//Q4 :
*ptr1 = 12.5;
//Q5 :
*ptr2 = 5.76;
//Q6 :
float variableIntermediaire;

variableIntermediaire = * ptr1;
*ptr1 = *ptr2;
*ptr2 = variableIntermediaire;
```

Exercice 3 (problème)

- Écrire une procédure "permute" qui permet de permuter les valeurs de deux variables entières et écrire un programme dans lequel on saisira deux nombres entiers avant de faire appel a cette procédure et d'afficher le contenue de ces variable afin de vérifier la permutation.

Exercice 4 (problème)

Pointeurs et références *(exercice wikipedia)*.

- Donner et expliquer le résultat de l'exécution du programme suivant :

Code : C

```
#include <stdio.h>
#define taille_max 5

void parcours(int *tab)
{
     int *q=tab;
```

```c
        do
        {
            printf("%d:%d\n", q-tab, *q-*tab);
        }
        while (++q-tab < taille_max);
    }

    void bizarre(int **copie, int *source)
    {
        *copie=source;
    }

    int main(void)
    {
        int chose[taille_max] = {1,3,2,4,5}, *truc;
        printf("chose : \n");

        parcours(chose);

        bizarre(&truc, chose);

        printf("truc : \n");

        parcours(truc);

        return 0;
    }
```

Tableaux unidimensionnels

Exercice 1

- Déclarer de façon automatique un tableau à 15 variables de type `double` . Et initialiser ce tableau avec des zéros.

Solution

Secret **(cliquez pour afficher)**

Code : C

```c
double tab[15];
int i;

for(i=0 ; i< 15 ; i++)
    tab[i] = 0;
```

Exercice 2 (problème)

- Utiliser l'allocation dynamique pour créer un tableau de 15 variables de type `float` . Initialiser ce tableau avec des zéros, puis libérer la mémoire allouée si l'allocation a été effectuée avec succès.

Exercice 3 (problème)

- Déclarer un double pointeur sur `float` , puis utiliser une fonction pour effectuer l'allocation dynamique d'un tableau de 15 variables de type `float` (de deux façon, par utilisation du retour de la fonction, puis par un passage par référence).
- Utiliser une fonction pour initialiser ce tableau avec des zéros si l'allocation a été effectuée avec succès.
- Puis libérer la mémoire allouée toujours si l'allocation a été effectuée avec succès.

Exercice 4 (problème)

- Ecrire de deux façons différentes, un programme qui vérifie sans utiliser une fonction de <string>, si une chaîne CH introduite au clavier est un palindrome:
- a) en utilisant uniquement le formalisme tableau
- b) en utilisant des pointeurs au lieu des indices numériques

Rappel : Un palindrome est un mot qui reste le même qu'on le lise de gauche à droite ou de droite à gauche.

Exemples :

PIERRE : n'est pas un palindrome

OTTO : est un palindrome

23432 : est un palindrome

Exercice 5 (problème)

- Écrire un programme qui lit deux tableaux A et B et leurs dimensions N et M au clavier et qui ajoute les éléments de B à la fin de A. Utiliser le formalisme pointeur à chaque fois que cela est possible.

Exercice 6 (problème)

- Écrire un programme qui demande à l'utilisateur de saisir la dimension N d'un tableau T du type `int` (dimension maximale: 50), remplit le tableau par des valeurs entrées au clavier et affiche le tableau.

- Calculer et afficher ensuite la somme des éléments du tableau.

- Copiez ensuite toutes les composantes strictement positives dans un deuxième tableau TPOS et toutes les valeurs strictement négatives dans un troisième tableau TNEG.

- Afficher les tableaux TPOS et TNEG.

Tableaux multidimensionnels

Exercice 1 (problème)

- Écrire un programme qui lit les dimensions L et C d'un tableau T à deux dimensions du type int (dimensions maximales: 50 lignes et 50 colonnes). Remplir le tableau par des valeurs entrées au clavier et afficher le tableau ainsi que la somme de chaque ligne et de chaque colonne en n'utilisant qu'une variable d'aide pour la somme.

Exercice 2 (problème)

- Écrire un programme qui lit un texte (sur plusieurs lignes) saisi par l'utilisateur (chaque ligne peut avoir un nombre maximal de 100 caractères). La fin de la saisie sera validée par l'entrée d'une ligne vide.

- Écrire une fonction qui demande à l'utilisateur de saisir un mot, elle recherchera ce mot dans le texte saisi (que nous lui passerons sous forme d'argument) et retourne le numéro de la ligne du premier mot trouvé dans le texte.

En conclusion nous allons donc retenir les choses suivantes :

- Un pointeur n'est pas un tableau.
- Un tableau n'est pas un pointeur.
- Un pointeur doit toujours être initialisé, soit par une allocation dynamique soit en pointant sur une variable.

Ayez également en esprit que les codes les plus simples sont souvent les plus robustes et fiables. Ne cherchez pas automatiquement la solution compliquée, appliquez ce que vous avez appris dans ce cours qui en fin de compte a été rédigé dans un but de vous montrer comment utiliser correctement chaque technique (d'allocation, libération, traitement d'erreur, initialisation...). Définissez vos contraintes de portabilité et de conformité à l'une des normes dés la rédaction de votre cahier des charges, et avancez dans le développement de vos programmes en respectant cette contrainte.

Ainsi vous amoindrirez le risque de bogue, et vous obtiendrez un programme fiable en gestion d'erreurs en ce qui concerne tableaux et pointeurs .

Merci pour votre attention, et n'hésitez pas à me faire part de vos remarques.