

# Allocation dynamique

L'objectif de ce cours est de maîtriser l'allocation dynamique et la gestion de la mémoire. Vous découvrirez dans un premier temps une autre section de la mémoire que vous n'avez pas encore manipulé jusqu'à présent : le tas. Par la suite, vous apprendrez à manipuler les fonctions d'allocation usuelles.

## 1 Pile ou Tas ?

Vous savez déjà que lorsque vous exécutez un programme, ce dernier trouve une place libre dans la mémoire vive (la RAM) de votre ordinateur. Afin de pouvoir s'exécuter correctement, et sans prendre de mémoire à d'autres programmes, chaque programme qui s'exécute se voit attribuer une certaine quantité de RAM. Cette quantité est appelée *pile* du programme.

Il n'y a aucune surprise avec le nom de cette mémoire. Vous savez déjà que les déclarations de variables se font successivement en mémoire. On peut considérer qu'elles s'empilent les unes sur les autres. Lorsqu'on quitte une fonction, les variables locales à cette fonction sont supprimées : la mémoire est libérée.

### TD Exercice 1.

Lorsque qu'un programme C demande explicitement un supplément de mémoire à l'aide des fonctions d'allocation dynamique (`malloc`, `calloc` ...), un autre type de mémoire est utilisé : le *tas*. Chaque programme peut réserver une partie de cet espace tant qu'il reste de la place.<sup>1</sup> La différence fondamentale avec la pile est que le programmeur doit gérer manuellement l'allocation et la libération de la mémoire sur le tas.

#### Différences entre la pile et le tas.

##### La pile :

- Allocation et désallocation automatique.
- Taille maximale fixée dès le lancement du programme.
- Impossible d'allouer une taille différente entre deux exécutions du même programme.

##### Le tas :

- Mémoire de grande capacité.
- Persistance des données entre les différentes portions de code.
- Besoin de sauvegarder l'adresse de l'emplacement réservé.
- Besoin de libérer manuellement la mémoire.

Notons également une différence fondamentale entre la pile et le tas. Lorsqu'une variable est allouée sur la pile, elle est disponible de façon locale. Autrement dit, la portée

1. En réalité, même s'il ne reste plus de place, on peut en réserver...

d'une variable déclarée sur la pile est entre les accolades entourant la variable. Lorsqu'un emplacement mémoire<sup>2</sup> est réservé sur le tas, il est possible d'atteindre cet emplacement de n'importe quel endroit du code, tant que l'on sait où chercher.

## 2 Allocation dynamique de tableaux

Voyons maintenant les fonctions de manipulation de la mémoire (sous-entendu, du tas). Toutes ces fonctions sont accessibles via la librairie `stdlib`.

```
void *calloc (size_t nmemb, size_t size); // Allocation + Mise à 0
void *malloc (size_t size);               // Allocation seule
void *realloc (void *ptr, size_t size);   // Re-allocation
void free (void * ptr);                   // Libération
size_t sizeof (type OU variable)         // Taille
```

Les trois premières fonctions permettent de réserver un espace mémoire sur le tas et retournent l'adresse de cet espace mémoire. La fonction `calloc` prend en paramètres le nombre d'éléments `nmemb` de taille `size` à allouer et renvoie un pointeur sur ces données mises à zéro. En cas d'échec, ce pointeur est nul. La fonction `malloc` réalise les mêmes actions, sauf la mise à zéro des éléments.


La fonction `realloc` prend en paramètre un pointeur `ptr` sur des données déjà allouées et modifie cette allocation. La nouvelle zone mémoire est alors de `size` octets. La fonction retourne elle aussi un pointeur sur cette zone mémoire.

Afin de libérer la mémoire, vous devez appeler la fonction `free` dès que les données ne sont plus utilisées. Elle libère simplement la mémoire désignée. Attention, le pointeur `ptr` n'est pas modifié, il pointe donc toujours vers la même zone mémoire. Or, après l'appel à `free`, cette zone mémoire n'est plus réservée. Il ne faut plus la modifier, et il vaut mieux alors mettre le pointeur `ptr` à `NULL` après tout appel à `free`.

Enfin l'opérateur `sizeof` prend en paramètre un type de donnée et renvoie sa taille. On peut donc trouver les types `int`, `char`, `int*` ... parmi les types envoyés à `sizeof`.

**Exemple 1.** Concrètement, l'allocation d'un tableau de 10 entiers nécessite donc la déclaration préalable d'un pointeur sur entier : `int *tab`. L'allocation du tableau est alors réalisée de la façon suivante.

```
tab = (int*) calloc (10, sizeof(int));
```

La manipulation du tableau est ensuite tout à fait classique et se réalise via le pointeur `tab`. Une fois qu'on est certain que plus accès au tableau n'est nécessaire, il faut libérer la mémoire à l'aide de l'instruction `free(tab)`; 

Précisons que les trois premières fonctions renvoient des pointeurs de type `void`. Il ne s'agit en aucun cas d'un pointeur sur rien ! Il s'agit d'un type générique, à partir duquel on peut facilement réaliser des casts. Afin d'accéder à la bonne case dans un tableau (et donc au bon octet)<sup>3</sup>, il est nécessaire de préciser un type pour le tableau. Lors de l'appel

2. Notez l'utilisation du mot *emplacement mémoire* et non *variable*.

3. Un tableau de type `int` aura des cases espacées de 4 octets, un tableau de type `short` aura des cases séparées de 2 octets etc.

d'une des trois fonctions présentées ci-dessous, on arrive alors sur une instruction de la forme suivante.

```
type *ptr = calloc (...);
```

Le pointeur `ptr` possède un type particulier, et la fonction d'allocation renvoie un pointeur de type `void`. Le cast de ce dernier est donc obligatoire si on veut coder proprement.

**Exemple 2.** La tableau ci-après présente l'état de la mémoire à la fin du code ci-dessous.

```
void foo (char *tab, char c )
{
    *tab = c;
}
void main()
{
    int i;
    char *tab = (char*)calloc(5, sizeof(char));
    char *str = "toto";
    for (i=0; i<5; i++)
        foo (tab+i, str[i]);
}
```

Pile			Tas		
Adresse	Valeur	Nom	Adresse	Valeur	Nom
0x0160	0 1 2 3 4 5	i	0x4AC0	0 't'	tab[0]
0x0161			0x4AC1	0 'o'	tab[1]
0x0162			0x4AC2	0 't'	tab[2]
0x0163			0x4AC3	0 'o'	tab[3]
0x0164	0x4AC0	tab	0x4AC4	0	tab[4]
0x0165			0x4AC5		
0x0166			0x4AC6		
0x0167			0x4AC7		
0x0168	't'	str[0]	0x4AC8		
0x0169	'o'	str[1]	0x4AC9		
0x016A	't'	str[2]	0x4ACA		
0x016B	'o'	str[3]	0x4ACB		
0x016C	'\0'	str[4]	0x4ACC		
0x016D	0x4AC0 0x4AC1	tab	0x4ACD		
0x016E	0x4AC2 0x4AC3		0x4ACE		
0x016F	0x4AC4		0x4ACF		
0x0170			0x4AD0		
0x0171	't' 'o' 't' 'o' '\0'	c	0x4AD1		

### 3 Allocation de matrices

Maintenant que vous maîtrisez l'allocation de tableaux, il est temps de passer à l'allocation de matrices. L'objectif de cette fin de chapitre est donc de développer un programme réalisant un pivot de Gauss. Cette réalisation passe tout d'abord par l'allocation dynamique de matrices.

Vous savez déclarer des matrices de façon *statique*, dont les dimensions ne s'adaptent donc pas à l'utilisateur. Dans notre contexte, l'utilisateur doit avoir une liberté totale sur les dimensions du système. Sachant qu'une matrice peut être vue comme un tableau de tableaux, les fonctions d'allocation précédentes vont nous être utiles.<sup>4</sup>

Voyons donc une matrice  $m \times n$  ( $m$  lignes,  $n$  colonnes) comme un ensemble de  $m$  lignes. Chaque ligne peut être allouée pour former un tableau de  $n$  cases. La matrice, du point de vue informatique, n'est donc qu'un ensemble composé de  $m$  tableaux de  $n$  cases. Un tableau est identifié à l'aide d'un pointeur sur sa première case. Donc notre ensemble de tableaux peut se résumer à un ensemble de  $m$  pointeurs.

Rappelons qu'un pointeur est une variable qui contient une adresse. Le type du pointeur correspond au type de donnée que l'on trouve à l'adresse pointée. Par exemple, un pointeur déclaré de la façon suivante : `short *ptr`; pointe sur une variable de type `short`. Notre matrice étant un tableau de pointeurs, ce tableau est de type `type **mat`, où `type` désigne le type de donnée des coefficients de la matrice. Le schéma 1 résume cette architecture pour une matrice entière de 3 lignes, 4 colonnes.

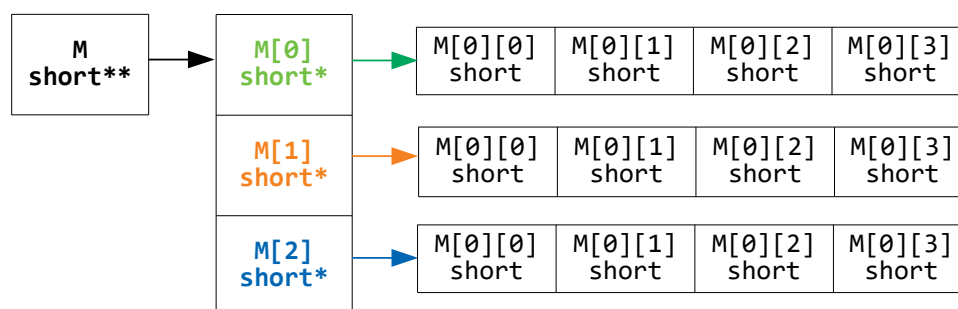


FIGURE 1 – Schéma d'une allocation dynamique de matrice  $3 \times 4$

Voyons maintenant comment réaliser ce genre d'allocation. En étudiant le schéma 1, on peut observer la présence d'un pointeur de type `short**`, d'un tableau de 3 cases contenant des `short*`, et de trois tableaux de 4 cases contenant des `short`.

Supposons que, comme sur la figure, on veuille allouer une matrice  $3 \times 4$  d'entiers nommée `M`. Le premier élément à allouer est le double pointeur. Parce qu'il est seul, une allocation statique suffit. Parce que vous savez que coder proprement, c'est important, vous n'oublierez pas d'initialiser le pointeur à `NULL`.

```
short **M = NULL;
```

Ce pointeur doit alors pointer sur un tableau de 3 pointeurs sur `short`. Une allocation dynamique est alors nécessaire. Le pointeur `M` étant de type `short**`, le cast est du même

4. C'est l'heure de respirer un grand coup !

type. Enfin, chaque case du tableau contenant un pointeur sur short, le type à placer dans le `sizeof` est `short*`.

```
M = (short**) calloc (3, sizeof(short*));
```

Il faut maintenant allouer les trois tableau d'entier courts de taille 4. L'adresse de chacun d'entre eux doit être stockée dans le tableau M. Une boucle est alors nécessaire. À chaque tour, une allocation de tableau 1D de type `short` est réalisée, est l'adresse est stockée dans `M[i]`.

```
for (i=0; i<3; i++)  
    M[i] = (short*) calloc (4, sizeof(short));
```

La partie pratique est maintenant terminée. Vous n'oublierez pas de vérifier chacune des allocations réalisées. Vous n'oublierez pas non plus de libérer toute la mémoire allouée. Sachant que l'allocation d'une matrice nécessite une boucle d'allocation, sa libération complète nécessite elle aussi une boucle. Cette libération vous est volontairement laissée en exercice.

**Exemple 3.** Le tableau 2 présente l'état de la mémoire à la fin du code ci-dessous.

```
int main()  
{  
    int i;  
    char **M = NULL;  
    M = (char**) calloc (3, sizeof(char*));  
    for (i=0; i<3; i++)  
        M[i] = (char*) calloc (2, sizeof(char));  
    M[1][0] = 'z';  
}
```

**TD** Exercice 4.

Pile			Tas		
Adresse	Valeur	Nom de var.	Adresse	Valeur	Nom de var.
0x0160	0 1 2	i	0x4AC0	0x4ACC	M[0]
0x0161			0x4AC1		
0x0162			0x4AC2		
0x0163			0x4AC3		
0x0164	0x4AC0	M	0x4AC4	0x4AD0	M[1]
0x0165			0x4AC5		
0x0166			0x4AC6		
0x0167			0x4AC7		
0x0168			0x4AC8	0x4AD4	M[2]
0x0169			0x4AC9		
0x016A			0x4ACA		
0x016B			0x4ACB		
0x016C			0x4ACC	0	M[0] [0]
0x016D			0x4ACD	0	M[0] [1]
0x016E			0x4ACE	0 'z'	M[1] [0]
0x016F			0x4ACF	0	M[1] [1]
0x0170			0x4AD0	0	M[2] [0]
0x0171			0x4AD1	0	M[2] [1]
0x0172			0x4AD2		
0x0173			0x4AD3		
0x0174			0x4AD4		
0x0175			0x4AD5		
0x0176			0x4AD6		
0x0177			0x4AD7		

FIGURE 2 – État de la mémoire après l'exécution du code de l'exemple 2.