



ECE PARIS • LYON
ÉCOLE D'INGÉNIEURS

Algorithmique et programmation structurée en C

Cours n°8

Langage C

Allocation dynamique et tableaux dynamiques

Antoine Hintzy



Plan du cours


- Allocation automatique (statique)
- Allocation dynamique
- Taille d'un type : `sizeof`
- Allocation dynamique de la mémoire : `malloc` / `calloc`
- Réallocation dynamique de la mémoire : `realloc`
- Libération de la mémoire : `free`
- Allocation automatique **vs** dynamique
- Différencier les deux allocations
- Tableau automatique / dynamique : comparaison
- Tableaux dynamiques à deux dimensions

Allocation automatique

Lorsque nous déclarons une variable en suivant la syntaxe `type nomDeLaVariable;`, un espace mémoire lui est alloué jusqu'à l'accolade fermante du ***bloc*** (***scope***) dans lequel elle a été déclarée.

On parle alors d'allocation **automatique**.

Remarque : L'allocation automatique est souvent appelée "allocation ***statique***" à tort ([en savoir plus](#)).

Les variables allouées automatiquement sont ajoutées dans ce qu'on !
appelle la **pile** d'exécution ( *call stack*).

Ce sont les dernières variables allouées qui sont libérées en premier.

Allocation automatique - Exemple

```
void uneFonctionOuProcedurePossiblementMain(int n) { // n existe dans tout le bloc  
!
```

```
    float a = 5.0f;        // a existe en mémoire à partir d'ici
```

```
    float* b = &a;         // b existe en mémoire à partir d'ici
```

```
    int tab[10];           // tab existe en mémoire à partir d'ici
```

```
    Point p;               // p existe en mémoire à partir d'ici
```

```
} // Libération automatique de n, a, b, tab et p (à la fin du scope/bloc).
```

n, a, b, tab et **p** sont des variables allouées automatiquement.

Allocation dynamique

Permet d'allouer et libérer des espaces mémoire quand on le souhaite.

C'est un peu de la **mémoire à la demande**. On indique expressément quand allouer et quand libérer.

Nous avons la **responsabilité de libérer la mémoire allouée dynamiquement**.

Les allocations dynamiques ne sont libérées qu'à notre demande. Elles survivent aux blocs (accolades).

⚠ Si on ne libère pas nos espaces mémoire alloués dynamiquement, rien ne le fera pour nous, il faudra redémarrer l'ordinateur pour retrouver les octets que nous n'avons pas libérés.

Les variables allouées dynamiquement sont ajoutées dans ce qu'on appelle le **tas** ( *heap*).

Il n'y a pas d'ordre précis dans la libération.

Allocation dynamique - Utilité

L'allocation dynamique est utilisée pour :

- ⦿ N'allouer que si nécessaire
- ⦿ Contrôler le cycle de vie (allocation/libération)
Choisir quand allouer et quand libérer, en plus d'avoir des données qui survivent aux blocs d'accolades.
- ⦿ Créer des tableaux de la bonne taille
Pas de case en trop.
- ⦿ Pouvoir redimensionner un tableau en fonction des nouveaux besoins
Ajouter/supprimer des cases.
- ⦿ Garantir une adresse fixe en mémoire, à condition de ne pas utiliser realloc

Allocation dynamique - Exemple

```
Point* uneFonctionOuProcEDUREPossiblementMain (int n) { // ● Allocation automatique de n (entier) ! !  
  
    float* a = (float*) malloc(sizeof(float)); // ● Allocation automatique de a (pointeur sur un float) / ● Allocation dynamique d'un float (4 octets)  
  
    *a = 5.0f; // on stocke 5.0f là où pointe a, donc dans les 4 octets du float alloué dynamiquement ci-dessus, puisque a pointe sur le float alloué dynamiquement  
  
    free(a); // ■ Libération du float dont l'adresse est stockée dans a. On ne peut plus utiliser l'espace mémoire pointé par a, mais le pointeur a existe encore lui !  
    a = NULL; // le pointeur a contient encore l'adresse du float qui a été libéré. On sécurise notre programme en faisant pointer a sur NULL afin qu'il ne contienne plus l'adresse qui ne nous appartient plus  
  
    int* tab = (int*) malloc(10 * sizeof(int)); // ● Allocation automatique de tab (pointeur sur entier) ● / Allocation dynamique de 10 entiers (10 x 4 = 40 octets)  
    Point* p = (Point*) malloc(sizeof(Point)); // ● Allocation automatique de p (pointeur sur un Point) ● / Allocation dynamique d'un Point (taille inconnue dans cet exemple)  
  
    free(tab); // ■ Libération des 10 entiers (40 octets) alloués dynamiquement, on ne peut plus les utiliser  
    tab = NULL; // facultatif : on pourrait ne pas remettre à NULL ici car de toute façon tab sera détruit juste après (la fonction se termine après cette instruction)  
  
    // ATTENTION : on remarque que le Point qu'on a alloué dynamiquement et dont l'adresse est stockée dans p n'est pas libéré, il continuera donc à exister en mémoire  
    // même après la fin de l'exécution de cette fonction, jusqu'à ce qu'on le libère avec free() ou qu'on redémarre l'ordinateur.  
  
    return p; // on retourne ici la valeur de p (avant sa destruction/libération). On retourne donc l'adresse du Point alloué dynamiquement. Ce Point est maintenant entre les mains du programme appelant !  
  
} // ■ Libération de n, a, tab et p
```

Le Point n'est pas libéré ! \$"

Légende :

- Allocation automatique, ■ Libération d'un espace mémoire alloué automatiquement
- Allocation dynamique, ■ Libération d'un espace mémoire alloué dynamiquement

Allocation dynamique

Mode d'emploi

- ⦿ Réserver un espace mémoire en précisant sa taille en octets

Avec `malloc()`/`calloc()` et `sizeof`

- ⦿ *Utiliser la mémoire allouée dynamiquement*

- ⦿ Libérer cet espace mémoire lorsqu'on n'en a plus besoin

Avec `free()`

Allocation dynamique

Connaitre la taille d'un type

Pour allouer dynamiquement de la mémoire, nous devons préciser combien d'octets nous souhaitons.

La fonction `sizeof()` prend en paramètre un type, et retourne sa taille en octets (au format `unsigned long`) :

```
sizeof(int) ; // retourne 4  
sizeof(char) ; // retourne 1  
sizeof(Point) ; // retourne la taille d'une structure Point
```

Connaitre la taille d'un type - Exemple avec une structure

Allocation dynamique

```
#include <stdio.h>

typedef struct {
    float x, y, z;
} Point;

int main(void) {
    printf("Taille d'un Point : %lu octets (= 3 fois la taille d'un float (%lu octets)).\n", sizeof(Point), sizeof(float));
    return 0;
}
```

⊙ Affiche :

Taille d'un Point : 12 octets (= 3 fois la taille d'un float (4 octets)).

⊙ `%lu` pour `unsigned long`, *fonctionne aussi avec `%d`*

Allocation dynamique avec **malloc/calloc**

Les deux **fonctions** permettant d'allouer dynamiquement de la mémoire sont **malloc()** et **calloc()**.

```
malloc(1 * sizeof(int)); // alloue un entier  
malloc(5 * sizeof(int)); // alloue un tableau de 5 entiers calloc(5, sizeof(int)); //
```

alloue un tableau de 5 entiers et les initialise à 0 On les utilise aussi bien pour allouer un seul élément ou plusieurs (tableaux dynamiques).

Leur utilisation nécessite l'inclusion de la bibliothèque standard **stdlib** :

```
#include <stdlib.h>
```

Allocation dynamique avec **malloc**

malloc() prend en paramètre le nombre d'octets à allouer, alloue cet espace mémoire et retourne son adresse.

Il faut alors stocker cette adresse dans un pointeur (généralement alloué automatiquement) et utiliser ce dernier comme un simple pointeur ou comme un tableau (qu'il faut initialiser nous-même (sinon, préférer **calloc()**)).

```
int* notes = (int*) malloc(6 * sizeof(int)); // Allocation dynamique de 6x4 octets
                                           // (la taille d'un tableau de 6 entiers)
                                           // dont l'adresse est affectée au pointeur notes

// NB : initialiser le tableau si besoin.
notes[2] = 12; // utilisation comme un tableau classique
// ... free(notes); // Libération de la mémoire allouée dynamiquement (obligatoire) notes = NULL; //
car après un free, le pointeur contient toujours l'adresse du bloc mémoire libéré !
// La variable automatique notes sera libérée automatiquement à la fin du bloc
```

🟡 Attention à ne jamais perdre l'adresse d'un emplacement mémoire alloué dynamiquement. Elle doit toujours être stockée dans un pointeur pour que l'on puisse libérer son "⚠️" espace mémoire.

😓 Bonne pratique : Le type de retour de **malloc** est **void*** (pointeur générique). Bien que l'ordinateur soit capable de *caster* implicitement un **void*** en **int*** (par exemple), il est préférable d'indiquer explicitement le cast :

(int*) malloc... (*évite les warning*).

Allocation dynamique avec **calloc**

calloc fonctionne comme **malloc** : cette fonction alloue un espace mémoire et retourne l'adresse de son premier octet. Elle se distingue cependant sur deux points :

- ⊙ **calloc** prend **2 paramètres** : le nombre d'éléments, puis le nombre d'octets de chaque élément.
- ⊙ **calloc** initialise tous les octets à 0 contrairement à **malloc** qui laisse la mémoire récupérée telle quelle.

```
int * notes = (int*) calloc(6, sizeof(int)); // Allocation dynamique de 6x4 octets, // soit la taille d'un tableau de 6 entiers  
  
(avec mise à 0 de tous les bits) notes[2] = 12; // utilisation comme un tableau classique  
  
// ...free(notes); // Libération de la mémoire allouée dynamiquement (obligatoire)  
notes = NULL;
```

Allocation dynamique avec **malloc/calloc**

Les deux fonctions **malloc** et **calloc** peuvent retourner **NULL** en cas d'erreur (pas assez d'espace mémoire...). **NULL** représente l'absence de valeur en **C**.

Il peut alors être intéressant de vérifier si la valeur retournée par **malloc** ou **calloc** est **NULL**.

```
Point* p = (Point*) malloc(sizeof(Point)); // allocation d'un unique Point (éq. à un tab d'une case)
if (p != NULL) {
    // le Point dynamique a été alloué, on l'utilise :
    p->x = 5;
    free(p); // puis on le libère éventuellement
    p = NULL;
}
else
{
    printf("L'allocation a échoué.\n");
}
```

Allocation dynamique avec **malloc/calloc**

On peut aussi utiliser **assert()** de la bibliothèque **assert.h** :

Cette fonction permet de "s'assurer que" avant de continuer.

```
#include <stdlib.h>
#include <assert.h>

int main(void) {
    Point* p = malloc(sizeof(Point));
    assert(p != NULL); // Si la condition est fausse, le programme entier sera interrompu
    // ...
    free(p);
    p = NULL;
    return 0;
}
```

Réallocation dynamique de la mémoire avec realloc

La fonction `realloc()` permet de **réallouer un bloc de mémoire alloué dynamiquement** (un tableau), notamment afin de l'**agrandir** ou le **rétrécir**. Il prend en paramètres l'adresse du bloc mémoire à réallouer, suivi de sa nouvelle taille souhaitée, puis retourne sa nouvelle adresse. En effet, s'il n'y a pas assez de place dans la mémoire à la suite du bloc déjà alloué pour l'agrandir, ce dernier est **déplacé à un nouvel endroit dans la mémoire**, permettant de stocker tous ses octets de façon contiguë.

● Il ne faut alors jamais faire pointer un pointeur sur la case d'un tableau dynamique si ce dernier est susceptible d'être réalloué (et donc déplacé).

```
int* p = (int*) malloc(sizeof(int)); // un entier, équivalent à un tableau d'un entier
assert(p != NULL);
*p = 4; // équivalent à p[0] = 4; car un tableau est un pointeur
p = realloc(p, 10 * sizeof(int)); // devient un tableau de 10 entiers
printf("%d\n", p[0]); // affiche 4
p[1] = 7;
// ...
free(p);
p = NULL;
```


Libération de la mémoire

La mémoire de l'ordinateur est partagée avec d'autres programmes. Nous ne pouvons pas nous permettre de la monopoliser !

Il faut impérativement **libérer les espaces mémoire que l'on a alloués**

dynamiquement (`malloc/calloc`) avec la fonction `free()` qui prend en paramètre l'adresse du bloc mémoire à libérer (on lui donne généralement la valeur du pointeur qui pointe dessus).

```
int* p = (int*) malloc(sizeof(int)); // allocation d'un entier
float* notes = (float*) malloc(10 * sizeof(float)); // allocation d'un tableau de 10 floats
...
free(p);
p = NULL;
free(notes);
notes = NULL;
```

Libération de la mémoire

⚠️ Tout espace mémoire alloué dynamiquement qui n'est pas libéré ⚠️ avec **free ()** ne sera récupérable et réutilisable qu'après la mise hors tension de l'ordinateur.

💡 Astuce : en écrivant votre code, ajoutez le **free** au même moment que **malloc/calloc** afin de ne pas l'oublier.

1 malloc/calloc = 1 free

Libération de la mémoire

NB : si plusieurs pointeurs pointent sur le même espace mémoire alloué dynamiquement, inutile d'effectuer le **free** plusieurs fois. Une seule fois suffit :

```
Point* p = malloc(sizeof(Point)) ;
Point* pointPrincipal = p;
// p et pointPrincipal pointent les deux sur le même espace mémoire alloué
dynamiquement free(p) ;
p = NULL;
```

Faire **free(pointPrincipal)** ; reviendrait à libérer une seconde fois l'espace mémoire.

Différencier les deux allocations

Lorsque l'on écrit :

```
int* pEntier = (int*) malloc(sizeof(int)) ;
```

Deux espaces mémoires sont alloués !

- ◉ **pEntier** est alloué automatiquement (type **int***) et sera libéré automatiquement à la fin du bloc.
- ◉ un entier est alloué dynamiquement par le **malloc**. Il sera libéré lorsqu'on le libérera avec **free**.

On aurait pu écrire cette instruction en deux temps :

```
int* pEntier = NULL; // allocation automatique du pointeur pEntier
pEntier = (int*) malloc(sizeof(int)); // allocation dynamique d'un entier + affectation
// de son adresse dans le pointeur pEntier
```

Différencier les deux allocations

Pointeur ne veut pas dire allocation dynamique, on utilisait les pointeurs pour les passages par adresse, sans allocation dynamique !

On pourrait faire une allocation dynamique sans pointeur :

```
malloc (sizeof (int) ) ;
```

Cela ne provoque aucune erreur de compilation. MAIS, il est impossible d'accéder à l'espace mémoire alloué après cette instruction puisque l'on ne récupère pas son adresse. On ne pourra pas le libérer...

- ⊙ Les espaces mémoire alloués **automatiquement** ont un **nom**, celui de la variable. Ils ont également une adresse.
- ⊙ Les espaces mémoire alloués **dynamiquement** n'ont **pas de nom**, ils ne sont pas directement reliés à une variable, mais ils ont une adresse qu'il faut absolument stocker quelque part -> dans un pointeur.

Exemple complet d'un code bien construit

```
#include <stdio.h>

Point* creerPoint(int x, int y, int z) {
    Point* p = (Point*) malloc(sizeof(Point));
    p->x = x;
    p->y = y;
    p->z = z;
    return p;
}

void afficherPoint(const Point* p) { // le Point pointé par p est devenu constant dans cette fonction, impossible de le modifier par erreur
    printf("Point(%d,%d,%d)", p->x, p->y, p->z);
}

void libererPoint(Point** ppP) { // on reçoit le pointeur du main par adresse pour pouvoir le mettre à NULL
    free(*ppP);    *ppP = NULL;
}

int main(void) { // le main se lit facilement
    Point* p1 = creerPoint(4, 2, 3);
    afficherPoint(p1);
    libererPoint(&p1); // Attention, on donne l'adresse du pointeur pour que libererPoint puisse le mettre à NULL
    return 0;
}
```

Tableaux automatiques / dynamiques - Comparaison

Nous souhaitons réaliser un programme qui calcule la moyenne de plusieurs notes d'un devoir surveillé.

◉ Allocation **automatique** : nous devons prévoir de la place pour suffisamment de notes à l'avance, en dur, dans le code :

```
#define MAX 100 // réservation de 100 cases (entiers : 400 octets), même si la classe comporte 15 étudiants
int main() {
    int notes[MAX]; // Ce tableau peut contenir au maximum 100 notes, et sera libéré à la fin du main()
```

Impossible d'avoir une classe de plus de 100 étudiants avec ce code...

Tableaux automatiques / dynamiques - Comparaison

Nous souhaitons réaliser un programme qui calcule la moyenne de plusieurs notes d'un devoir surveillé.

- ⊙ Allocation **dynamique** : nous pouvons allouer un tableau dynamique d'entiers avec une taille choisie une fois le programme lancé :

```
int main() {  
    int nbNotes = 0;  
    scanf("%d", &nbNotes);  
    int* notes = (int*) malloc(nbNotes * sizeof(int)); // Allocation dynamique d'un tableau de `nbNotes` notes (entiers)  
    // utilisation du tableau...  
    free(notes); // libération de la mémoire à la demande (obligatoire car non-automatique)  
    notes = NULL;  
}
```

Tableaux : Allocation automatique ou dynamique ?

- ⊙ Allocation **automatique** : la taille des tableaux est définie "*en dur*", c'est-à-dire dans le code source (la taille maximale est donc **constante**).
- ⊙ C'est à nous de prévoir et de réserver suffisamment de cases en mémoire à l'avance, en définissant une taille souvent plus grande que nécessaire (taille physique).
- ⊙ Allocation **dynamique** : la taille des tableaux est choisie une fois le programme lancé, selon nos besoins à l'instant t .
- ⊙ Possibilité de ne réserver que le nombre de cases nécessaires.
- ⊙ Possibilité de ne réserver de la mémoire que si besoin ou encore à certains moments seulement.
- ⊙ Possibilité d'ajuster la taille du tableau en ajoutant ou supprimant des cases

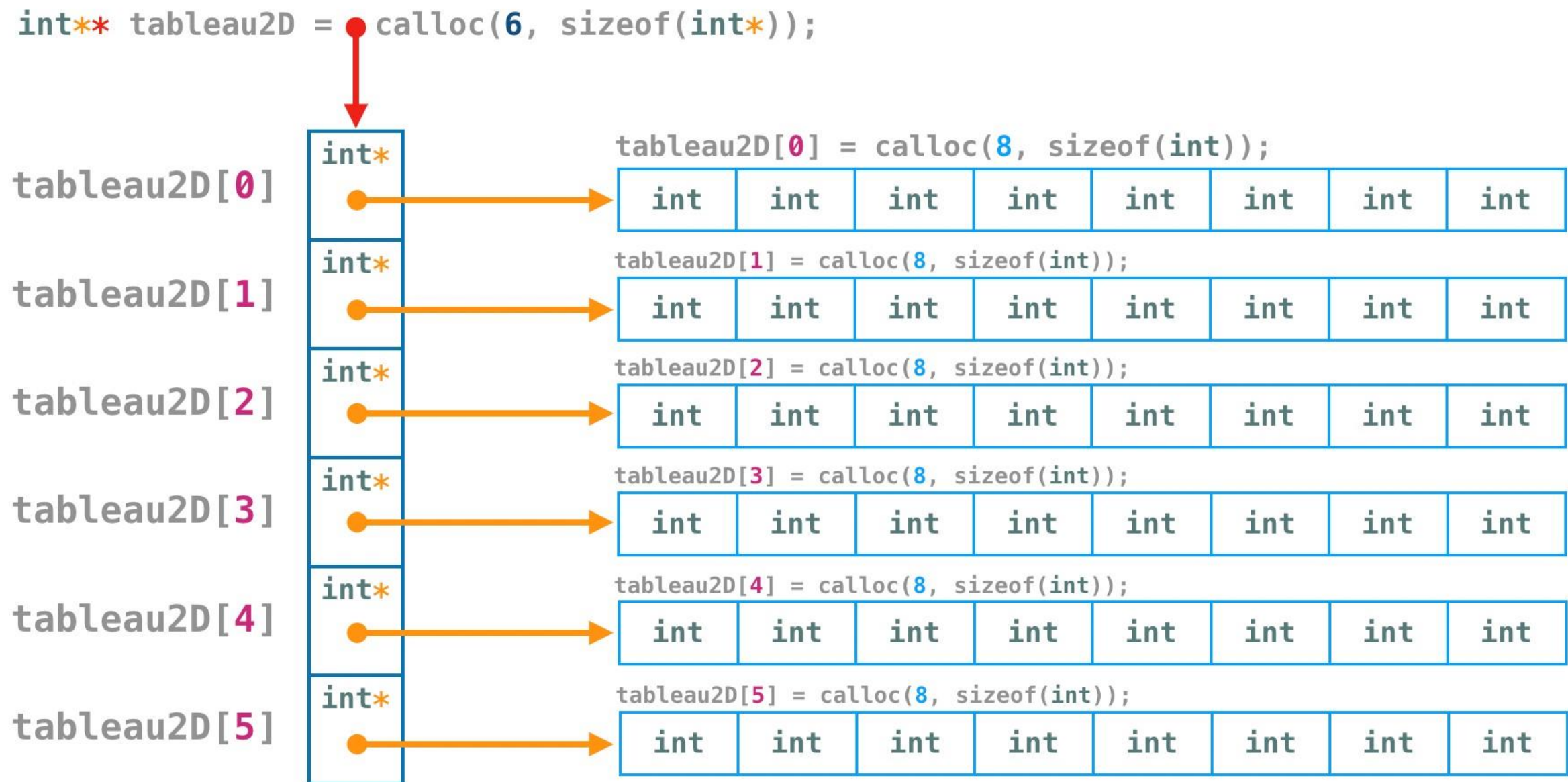
Tableaux dynamiques à deux dimensions

En allocation dynamique, un tableau à deux dimensions (*matrice*), se déclare par un double pointeur :

```
int** tableau2D; // tableau d'entiers à deux dimensions
```

Un tableau à deux dimensions est un tableau de pointeurs sur tableau. Ses éléments sont donc tous des tableaux de même type (`int` par exemple).

Tableaux dynamiques à deux dimensions



Tableaux dynamiques à deux dimensions - Allocation

```
int** allouerTableau2D(int nbLignes, int nbColonnes) {  
    int** tableau2D = NULL;  
    int i = 0;  
    tableau2D = (int**) calloc(nbLignes, sizeof(int*));  
    for(i = 0; i < nbLignes; i++) {  
        tableau2D[i] = (int*) calloc(nbColonnes, sizeof(int));  
    }  
    return tableau2D;  
}
```

Tableaux dynamiques à deux dimensions - Libération

```
void libererTableau2D(int** tableau2D, int nbLignes, int nbColonnes) {  
    int i = 0;  
    if (nbColonnes > 0) {  
        for (i = 0; i < nbLignes; i++) {  
            free(tableau2D[i]);  
        }  
    }  
    if (nbLignes > 0) {  
        free(tableau2D);  
    }  
}
```

● Un **free** par **malloc** / **calloc** !