



# **Allocation dynamique de mémoire**

# 1. Trouvez la taille d'une variable en fonction de son type

Selon le type de variable que vous demandez de créer, vous avez besoin de plus ou moins de mémoire. Le problème, c'est que l'espace pris en mémoire dépend des machines : peut-être que chez vous un `int` occupe 8 octets.

Pour vérifier quelle taille occupe chacun des types sur votre ordinateur, nous allons utiliser l'opérateur **`sizeof()`**.

Pour connaître la taille d'un **`int`**, on écrit :

**`sizeof(int)`**

À la compilation, cela sera remplacé par un nombre : le nombre d'octets que prend **`int`** en mémoire.



Affichant la valeur à l'aide d'un **printf**

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

int main(int argc, char *argv[])
{
    printf("char : %d octets\n", sizeof(char));
    printf("int : %d octets\n", sizeof(int));
    printf("long : %d octets\n", sizeof(long));
    printf("double : %d octets\n", sizeof(double));
    return 0;
}
```

Peut-on afficher la taille d'un type personnalisé qu'on a créé (une structure) ?

**sizeof** marche aussi sur les structures :

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

typedef struct Coordonnees Coordonnees;
struct Coordonnees
{
    int x;
    int y;
};

int main(int argc, char *argv[])
{
    printf("Coordonnees : %d octets\n", sizeof(Coordonnees));

    return 0;
}
```

**Remarque:** Plus une structure contient de sous-variables, plus elle prend de mémoire.

si on déclare une variable de type **int** :

**int nombre = 18;**

et que **sizeof(int)** indique 4 octets, alors la variable occupera 4 octets en mémoire !


Supposons que la variable nombre soit allouée à l'adresse 1600 en mémoire. On aurait alors :

Adresse	Valeur
1599	...
1600	18
1601	
1602	
1603	
1604	...

Notre variable **nombre** de type **int** qui vaut 18 occupe 4 octets dans la mémoire. Elle commence à l'adresse 1600 et termine à l'adresse 1603. La prochaine variable ne pourra donc être stockée qu'à partir de l'adresse 1604 !

Si on avait fait la même chose avec un **char**, on n'aurait occupé qu'un seul octet en mémoire :

Adresse	Valeur
1599	...
1600	18
1601	...
1602	...
1603	...
1604	...



Imaginez maintenant un tableau de **int** ! Chaque case du tableau occupera 4 octets. Si notre tableau fait 100 cases :

**Int tableau[100];**

on occupera alors en réalité  $4 * 100 = 400$  octets en mémoire.

Même si le tableau est vide, il prend 400 octets.

➡ Notez que si on crée un tableau de type **Coordonnees** :

**Coordonnees tableau[100];**

on utilisera cette fois :  $8 * 100 = 800$  octets en mémoire.





## 2. Allouez manuellement de la mémoire au système

Commencez par inclure la bibliothèque `<stdlib.h>`. Elle contient deux fonctions dont nous allons avoir besoin :

1. **malloc** (pour "Memory Allocation" ou allocation de mémoire, en français) : elle demande au système d'exploitation la permission d'utiliser de la mémoire.
2. **free** (libérer, en français) : elle indique au système que l'on n'a plus besoin de la mémoire qu'on avait demandée. La place en mémoire est libérée, un autre programme peut maintenant s'en servir au besoin.

Pour faire une allocation manuelle de mémoire, vous devez toujours suivre ces trois étapes :

1. Appeler **malloc** pour demander de la mémoire.
2. Vérifier la valeur retournée par malloc pour savoir si le système a bien réussi à allouer la mémoire.
3. Libérer la mémoire avec **free** une fois qu'on a fini d'utiliser la mémoire. Si on ne le fait pas, on s'expose à des fuites de mémoire, c'est-à-dire que votre programme risque de prendre beaucoup de mémoire alors qu'il n'a en réalité plus besoin de tout cet espace.



## ➤ Étape 1 : Demandez une allocation de mémoire avec malloc

Voici le prototype de la fonction **malloc**:

**void\* malloc(size\_t nombreOctetsNecessaires);**

La fonction prend en paramètre le nombre d'octets à réserver. Il suffit donc d'écrire **sizeof(int)** dans ce paramètre pour réserver suffisamment d'espace pour stocker un **int**.


**la fonction renvoie : un void\* !**

Dans le chapitre sur les fonctions, je vous avais dit que **void** signifiait "vide", et qu'on utilisait ce type pour indiquer que la fonction ne retournait aucune valeur. Alors ici, on aurait une fonction qui retourne un "pointeur sur vide" ?

En fait, cette fonction renvoie un pointeur indiquant l'adresse que le système a réservée pour votre variable. Si le système a trouvé de la place pour vous à l'adresse 1600, la fonction renvoie donc un pointeur contenant l'adresse 1600.

**Remarque:** Comme **malloc** ne sait pas quel type elle doit retourner, elle renvoie le type **void\*** . Ce sera un pointeur sur n'importe quel type.

On peut dire que c'est un pointeur universel.



Si je veux créer manuellement une variable de type **int** en mémoire, je devrai indiquer à **malloc** que j'ai besoin de **sizeof(int)** octets en mémoire. Je récupère le résultat du **malloc** dans un pointeur sur **int**:

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

int main(int argc, char *argv[])
{
    int* memoireAllouee = NULL; // on crée un pointeur sur int
    memoireAllouee = malloc(sizeof(int));

    return 0;
}
```

**memoireAllouee** est un pointeur contenant une adresse qui vous a été réservée par le système, par exemple l'adresse 1600 pour reprendre mes schémas précédents.

## ➤ Étape 2 : Testez le pointeur pour vérifier la valeur retournée par malloc

La fonction **malloc** a donc renvoyé dans notre pointeur **memoireAllouee** l'adresse qui a été réservée pour vous en mémoire. Deux possibilités :

1. Si l'allocation a marché, notre pointeur contient une adresse.
2. Si l'allocation a échoué, notre pointeur contient l'adresseNULL.

Il est peu probable qu'une allocation échoue, mais ça peut arriver : si vous demandez à utiliser 34 Go de mémoire vive, il y a peu de chances que le système vous réponde favorablement.

Si l'allocation a échoué, c'est qu'il n'y avait plus de mémoire libre (c'est un cas critique). Dans un tel cas, le mieux est d'arrêter immédiatement le programme parce que, de toute manière, il ne pourra pas continuer convenablement.

**Remarque:** La fonction `exit()` arrête immédiatement un programme. Elle prend en paramètre la valeur que le programme doit retourner.

Si le pointeur est différent de **NULL** , le programme peut continuer, sinon il faut afficher un message d'erreur, ou même mettre fin au programme.

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

int main(int argc, char *argv[])
{
    int* memoireAllouee = NULL; // on crée un pointeur sur int
    memoireAllouee = malloc(sizeof(int));
    if(memoireAllouee == NULL){ // si l'allocation a échoué
        exit(0); // On arrête immédiatement le programme
    }
    return 0;
}
```

### ➤ Étape 3 : Libérez de la mémoire avec free

Tout comme on utilisait la fonction `fclose` pour fermer un fichier dont on n'avait plus besoin, on va utiliser la fonction `free` pour libérer la mémoire dont on ne se sert plus.

La fonction `free` a juste besoin de l'adresse mémoire à libérer. On va donc lui envoyer notre pointeur : **memoireAllouee** dans notre exemple :

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

int main(int argc, char *argv[])
{
    int* memoireAllouee = NULL; // on crée un pointeur sur int
    memoireAllouee = malloc(sizeof(int));
    if(memoireAllouee == NULL){ // si l'allocation a échoué
        exit(0); // On arrête immédiatement le programme
    }

    // On peut utiliser ici la mémoire
    free(memoireAllouee); // On n'a plus besoin de la mémoire, on la libère

    return 0;
}
```

## ► Analysez un exemple concret d'utilisation

Demander l'âge de l'utilisateur et le lui afficher. La seule différence avec ce qu'on faisait avant, c'est qu'ici la variable va être allouée manuellement :

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

int main(int argc, char *argv[])
{
    int* memoireAllouee = NULL;

    memoireAllouee = malloc(sizeof(int)); // Allocation de la mémoire
    if (memoireAllouee == NULL)
    {
        exit(0);
    }

    // Utilisation de la mémoire
    printf("Quel age avez-vous ? ");
    scanf("%d", memoireAllouee);
    printf("Vous avez %d ans\n", *memoireAllouee);

    free(memoireAllouee); // Libération de mémoire

    return 0;
}
```



**Bref** : on y a alloué dynamiquement une variable de type **int**.

Ce qu'on a écrit revient finalement au même que d'utiliser la méthode automatique :

```
int main(int argc, char *argv[])
{
    int maVariable = 0; // Allocation de la mémoire (automatique)

    // Utilisation de la mémoire
    printf("Quel age avez-vous ? ");
    scanf("%d", &maVariable);
    printf("Vous avez %d ans\n", maVariable);

    return 0;
} // Libération de la mémoire (automatique à la fin de la fonction)
```

### Remarque:

En résumé, il y a deux façons de créer une variable, c'est-à-dire d'allouer de la mémoire :

1. Automatiquement : c'est la méthode que vous connaissez et qu'on a utilisée jusqu'ici.
2. Manuellement : c'est la méthode que je vous enseigne dans ce chapitre.



## ➤ Créez un tableau dont la taille n'est connue qu'à l'exécution

On se sert de l'allocation dynamique pour créer un tableau dont on ne connaît pas la taille avant l'exécution du programme.

Imaginons par exemple un programme qui stocke l'âge de tous les amis de l'utilisateur dans un tableau. Vous pourriez créer un tableau de **int** pour stocker les âges, comme ceci :

```
Int ageAmis[15];
```

Mais qui vous dit que l'utilisateur a 15 amis ? Peut-être qu'il en a plus que ça !

Comme je vous l'ai appris, il est interdit en C de créer un tableau en indiquant sa taille à l'aide d'une variable :

```
Int amis[nombreAmis];
```

Ce code fonctionne peut-être sur certains compilateurs, mais uniquement dans des cas précis, il est recommandé de ne pas l'utiliser !

L'allocation dynamique permet de créer un tableau qui a exactement la taille de la variable **nombreDAmis**, et cela grâce à un code qui fonctionnera partout !

- 
- Demandons au **malloc** de nous réserver **nombreAmis \* sizeof(int)** octets en mémoire : **amis = malloc(nombreAmis \* sizeof(int))**

Ce code permet de créer un tableau de type **int** qui a une taille correspondant exactement au nombre d'amis !



```
int main(int argc, char *argv[])
{
    int nombreDAmis = 0, i = 0;
    int* ageAmis = NULL; // Ce pointeur va servir de tableau après l'appel du malloc

    // On demande le nombre d'amis à l'utilisateur
    printf("Combien d'amis avez-vous ? ");
    scanf("%d", &nombreDAmis);

    if (nombreDAmis > 0) // Il faut qu'il ait au moins un ami (je le plains un peu sinon :p)
    {
        ageAmis = malloc(nombreDAmis * sizeof(int)); // On alloue de la mémoire pour le tableau
        if (ageAmis == NULL) // On vérifie si l'allocation a marché ou non
        {
            exit(0); // On arrête tout
        }

        // On demande l'âge des amis un à un
        for (i = 0 ; i < nombreDAmis ; i++)
        {
            printf("Quel age a l'ami numero %d ? ", i + 1);
            scanf("%d", &ageAmis[i]);
        }

        // On affiche les âges stockés un à un
        printf("\n\nVos amis ont les ages suivants :\n");
        for (i = 0 ; i < nombreDAmis ; i++)
        {
            printf("%d ans\n", ageAmis[i]);
        }

        // On libère la mémoire allouée avec malloc, on n'en a plus besoin
        free(ageAmis);
    }

    return 0;
}
```