INFO2009 - Introduction à l'informatique

Allocation dynamique

Bertrand Alexis

Université de Liège

2025

L'allocation dynamique est un moyen de demander, lors de l'exécution de votre programme, de réserver un certain espace dans la mémoire de votre machine. Faire ceci est notamment utile si vous voulez créer des tableaux ou passer des types structurés en dehors de la fonction qui l'utilise. Nous allons par la suite expliquer ces deux cas.

Tableaux

Le langage C est un langae compilé. Avant de pouvoir lancer votre programme, votre compilateur va transformer votre code en langage machine. Quand il fait ceci, il regarde en même temps l'espace mémoire nécessaire pour vos variables.
Lorsque vous créez des tableaux, vous mettez les valeurs des éléments dans des cases mémoires. Si vous avez un tableau avec une taille déjà définie lors de la compilation, le compilateur sait déjà le nombre de cases à réserver mais quand ce n'est pas le cas, que la taille peut différer en fonction des exécutions du programme ou que la taille est une variable, le compilateur ne peut pas savoir quoi réserver. C'est à ce moment là que nous devons réaliser une allocation dynamique.

Exemples

```
int t[4];
```

Cet exemple est permis et va réserver 4 cases mémoires qui se suivent pour ce tableau.

```
scanf("%d", &n);
int t[n];
```

Il n'est pas permis de faire ceci. Le compilateur ne connait pas n et ne pourra pas donc réserver le nombre de cases correct.

Exemples

```
int n = 4;
int t[n];
```

Il n'est pas permis de faire ceci. Même si on connait la valeur de n à la compilation, le compilateur n'est pas assez intelligent pour comprendre que n vaut 4 dans la deuxième ligne.

```
#define n 4
int t[n];
```

Ceci est permis et fonctionnera car lorsqu'on réalise un `define`, le compilateur va modifier toutes les occurences de n par 4.

Types structurés

Comme vous le savez, les variables sont locales aux fonctions les initialisent ou dont elles sont les arguments et disparaissent donc lorsque la fonction se termine. Lorsqu'on utilise des types structurés, la place prise n'est pas connuee comme les types standards (e.g. `int`, `float`, `char`,...). Afin de retourner une structure complète, nous devons le faire via un pointeur vers l'adresse mémoire où la structure est enregistrée. Cet espace doit donc être réservé dynamiquement.
Attention que si vous avez des tableaux qui n'ont pas une taille constante pour chaque exécution ou que vous avez des pointeurs dans le champs du type, vous devez également allouer dynamiquement ces champs en plus du type structuré de base.

Exemples

```
typedef struct {
    int *t;
} p;
void f(...) {
    p *n;
    n = malloc(sizeof(p));
    n->t = malloc(sizeof(int));
}
```

Dans cet exemple, on voit qu'on doit allouer les 2 pointeurs.

Exemples

```
typedef struct {
    int *t;
} p;
void f (...) {
    int *x;
    p *n;
    n = malloc(sizeof(p));
    x = malloc(sizeof(int));
    n->t = x;
}
```

Ici, j'ai alloué la variable x avant de la mettre dans le champs t de n. Je ne dois pas allouer $n->t$ car x a été alloué. Ça fonctionne car en réalité x et $n->t$ sont des adresses et c'est l'endroit pointé qui est alloué.

Fuite de mémoire

Comme vous réserver de la mémoire, une fois que vous n'en avez plus besoin, vous devez préciser que l'espace peut être ré-utilisé pour autre chose. C'est ce qu'on appelle libérer l'espace mémoire. Pour être sûr de ne pas avoir de fuites mémoire, il y a une règle simple à appliquer : 1 `malloc` = 1 free. (Attention, le free et le malloc ne sont pas obligés d'être dans la même fonction)
Attention, quand vous libérer l'espace mémoire, vous n'avez plus accès à ce qui est stocké et l'adresse n'est pas réinitialisée à `NULL`.

Exemples

```
void l(p *n) {
    free(n->t);
    free(n);
}
```

Ici, comme j'ai un champs qui a été alloué, je dois aussi le libérer, avant de libérer la variable du type structuré.

Exemples

```
typedef struct {
    int *t;
} p;

typedef struct p_t {
    struct p_t *next;
} p;

void l(p *n) {
    for (;*n; n = n->next) {
        free(n);
    }
}
```

Ceci n'est pas permis car en faisant `free(n)`, on oublie *n* et on

Exemples

ne peut pas récupérer `next`. Une manière de le faire serait :

```
void l(p *n) {
    p *suiv;
    for (;*n; n = suiv) {
        suiv = n->next;
        free(n);
    }
}
```