

Allocation Dynamique

L. Mounier

UGA - M2 CCI - PL1

22 octobre 2025

Rappels sur les principaux points abordés en C

■ Types

- types prédéfinis : entiers (dont booléens et caractères), réels
- constructeurs de types :
 - types énumérés, structures (produit de types)
 - tableaux (statiques), pointeurs**

■ Fonctions et “procédures”

- paramètres (donnée et résultat)
- **portée et durée de vie des données**

→ Utilisation de données **statiques** :

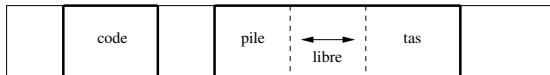
↪ taille, adresse mémoire et durée de vie **fixées à la compilation**

Allocation dynamique :

↪ *gestion à l'exécution* de la mémoire utilisée ...

Organisation (simplifiée) de la mémoire

Programme à l'exécution $\Rightarrow$ 3 zones mémoires distinctes



- le code : inst. du programme (en langage machine), taille fixe
- la pile (*stack*) : variables locales et paramètres
 $\hookrightarrow$ varie à chaque appels/retours de fonctions
- le tas (*heap*) : dédié aux allocations dynamiques
 $\hookrightarrow$ varie sur appels (implicite/explicite) de primitives systèmes

Bref rappel sur les pointeurs en C ...

Constructeur de type “pointeur sur un type T” (noté T*)

- représente l'**adresse (mémoire)** d'un élément de type T
- deux opérateurs associés :
 - déréférencement (*) = accès à l'élément “pointé”
p pointeur sur T $\Rightarrow$ *p est une valeur de type T
 - adresse (&) = accès à l'adresse d'une variable
x variable de type T $\Rightarrow$ &x est de type pointeur sur T

```
int a, b ; // variables entieres
int *p, *q ; // variables de type pointeur sur entier
int T[10] ; // tableau de 10 entiers
p = &a ; // p pointe sur la variable a
*p = 5 ; // affecte la valeur 5 a la variable a
q = p ; // p pointe egalement sur la variable a
b = *q // affecte 5 à la variable b
p = T ; // p pointe sur T[0]
```

Bref rappel sur les pointeurs en C ...

Constructeur de type “pointeur sur un type T” (noté T*)

- représente l'**adresse (mémoire)** d'un élément de type T
- deux opérateurs associés :
 - déréférencement (*) = accès à l'élément “pointé”
p pointeur sur T $\Rightarrow$ *p est une valeur de type T
 - adresse (&) = accès à l'adresse d'une variable
x variable de type T $\Rightarrow$ &x est de type pointeur sur T

```
int a, b ; // variables entieres
int *p, *q ; // variables de type pointeur sur entier
int T[10] ; // tableau de 10 entiers
p = &a ; // p pointe sur la variable a
*p = 5 ; // affecte la valeur 5 a la variable a
q = p ; // p pointe egalement sur la variable a
b = *q // affecte 5 à la variable b
p = T ; // p pointe sur T[0]
```

Primitives d'allocation dynamique en C

Allocation d'une zone en mémoire

```
(void *) malloc (int t)
```

- alloue (dans le tas) un bloc mémoire de taille `t` et renvoie un pointeur sur ce bloc ; renvoie `NULL` si l'allocation échoue
- la fonction `sizeof()` renvoie la taille (en octet) d'un type ou d'une variable

Libération d'une zone en mémoire

```
void free (void *p)
```

- libère le bloc mémoire pointée par `p`
- comportement indéfini si `p` ne pointe pas sur un bloc alloué

Ces primitives sont fournies par la librairie `<stdlib.h>`
(voir également `man malloc`).

Primitives d'allocation dynamique en C

Allocation d'une zone en mémoire

```
(void *) malloc (int t)
```

- alloue (dans le tas) un bloc mémoire de taille `t` et renvoie un pointeur sur ce bloc ; renvoie `NULL` si l'allocation échoue
- la fonction `sizeof()` renvoie la taille (en octet) d'un type ou d'une variable

Libération d'une zone en mémoire

```
void free (void *p)
```

- libère le bloc mémoire pointée par `p`
- comportement indéfini si `p` ne pointe pas sur un bloc alloué

Ces primitives sont fournies par la librairie `<stdlib.h>`
(voir également `man malloc`).

Exemple 1

```
void f(){
    int *p, *q ;
    *p = 12 ; // ERREUR : p contient pas une @ memoire valide
    p = (int *) malloc(sizeof (int)) ; // conversion de type
    *p = 12 ;    // OK
    q = p ;      // p et q pointent sur le meme bloc
    free (q) ;   // liberation du bloc pointe par p et q
}

int main(){
    f() ;
    return 0 ;
}
```


Exemple 2

Durée de vie d'un bloc alloué dans le tas

```
(char *) g() { // renvoie un pointeur sur char
    char *p ;
    p = (char *) malloc(sizeof (char)) ; // conversion de type
    return p ; // le bloc pointe par p RESTE ALLOUE
}
```

```
int main(){
    int *q ;
    q = g() ;
    *q = 'A' ; // q contient une @ memoire valide
    free (q) ; // liberation du bloc alloue par g()
    return 0 ;
}
```

Exemple 3

Durée de vie des variables locales

```
(char *) g() { // renvoie un pointeur sur char
    char *p ;
    char c ; // variable locale
    p = &c ; // OK, p contient l'adresse de c
    return p ; // PB, renvoie une @ qui va etre INVALIDE
}
```

```
int main(){
    int *q ;
    q = g() ;
    *q = 'A' ; // q contient une @ memoire INVALIDE
    return 0 ;
}
```

Application 1 : tableaux dynamique

La fonction `malloc()` permet de choisir la **taille** du bloc à allouer
→ **tableaux dynamiques** ...

Exemple

```
int *t ;

/* allocation d'un bloc de taille 10 entiers
t = (int *) malloc (10*sizeof(int)) ;

/* acces indexe possible */
t[0] = 4 ;
t[8] = 5 ;
free(t) ;
```

Variante : fonction `calloc()`

```
t = (int *) calloc (10,sizeof(int)); // alloue un bloc de 10 int
```

Application 2 : séquences chaînées

Représentation d'une séquence S en langage C ?

1 tableaux statiques :

- taille mémoire et durée de vie S fixée à la compilation
- aucun surcoût à l'exécution

2 tableaux dynamique :

- taille mémoire maximale de S fixée à l'allocation
- durée de vie de S peut varier à l'exécution
- (léger) surcoût à l'exécution (1 appel à allouer/libérer)

comment adapter **dynamiquement** la taille mémoire de S ?
(en fonction des insertions/suppressions d'éléments)

→ **séquences chaînées !**

Séquences chaînées en C

Notation algorithmique

Cellule : un type

Cellule : le type <elem: un entier, suiv: pointeur sur Cellule>

Seq : le type pointeur sur Cellule

Implémentation en C

```
typedef struct cellule {  
    int elem ;  
    struct cellule *suiv ;  
} Cellule ;
```

```
typedef Cellule *Seq ;
```

Le pointeur NULL représente la séquence vide

Séquences chaînées en C

Notation algorithmique

Cellule : un type

Cellule : le type <elem: un entier, suiv: pointeur sur Cellule>

Seq : le type pointeur sur Cellule

Implémentation en C

```
typedef struct cellule {  
    int elem ;  
    struct cellule *suiv ;  
} Cellule ;
```

```
typedef Cellule *Seq ;
```

Le pointeur NULL représente la séquence vide

Parcours d'une séquence chaînée

Incrémenter tous les éléments de la séquence S ?

```
void IncElements (Seq S) {  
    Seq elemC ; // element courant pour le parcours  
  
    elemC = S ;  
    while (elemC != NULL) { // tant qu'il existe encore un element  
        elemC->elem = elemC->elem + 1 ; // on l'incmente  
        elemC = elemC->suiv ; // on passe au suivant  
    }  
}
```

Le paramètre S est ici de type Seq car il n'est pas modifié
(passage par donnée)

Par contre chaque élément de S est modifié ...

Insertion en tête dans une séquence chaînée (1)

Version 1 : fonction

`Seq InsTete (Seq S, int x)`
ajoute le nouvel élément `x` en tête de `S` et renvoie la nouvelle séquence

```
Seq InsTete (Seq S, int x) {  
    Seq tmp ; // temporaire pour le nouvel element  
  
    // allocation d'un nouvel element x  
    tmp = (Seq) malloc (sizeof(Cellule)) ;  
    tmp->elem = x ;  
    tmp->suiv = S ; // chainage en tete de S  
    return tmp ; // renvoie la nouvelle sequence  
}
```


Insertion en tête dans une séquence chaînée (2)

Version 2 : action avec paramètre résultat

```
void InsTete (Seq *S, int x)
```

modifie S en lui ajoutant en tête un nouvel élément x

```
void InsTete (Seq *S, int x) {
```

```
    Seq tmp ; // temporaire pour le nouvel element
```

```
    // allocation d'un nouvel element x
```

```
    tmp = (Seq) malloc (sizeof(Cellule)) ;
```

```
    tmp->elem = x ;
```

```
    tmp->suiv = *S ; // chainage en tete de S
```

```
    *S = tmp ; // affecte a S ce nouveau (premier) element
```

```
}
```

Le paramètre S est ici de type Seq * car il va être modifié (passage par donnée-résultat)

Et dans d'autres langages ?

Allocation dynamique présente dans la plupart des langage de programmation, mais sous différentes formes . . .

- C, C++

- gestion mémoire entièrement à la charge du programmeur
- risque important d'erreurs
- surcoût réduit (en temps d'exécution, en mémoire utilisée)

- Java

- gestion mémoire partiellement à la charge du programmeur
libération "automatique" (*ramasse-miettes*)
- peu de risque d'erreurs
- surcoût possible (en temps d'exécution, en mémoire utilisée)

- JavaScript, Python

- gestion mémoire implicite (allocation, libération)
- peu de risque d'erreurs
- surcoût (en temps d'exécution, en mémoire utilisée)