Allocation dynamique en C

Où les tableaux et les pointeurs finissent par se rejoindre…

Nicolas Gazères

Dassault Systèmes
DS Research, Life Sciences
ngs@3ds.com

Allocation dynamique

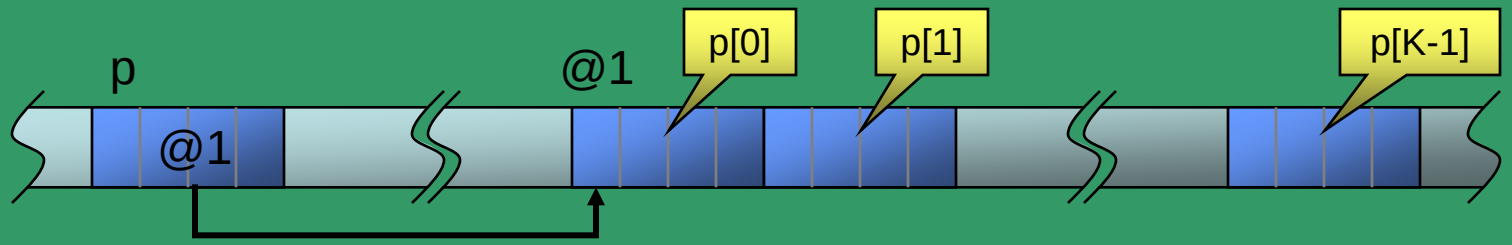
- Il existe des besoins en mémoire dont la taille est inconnue à la compilation et n'est connue qu'à l'exécution.
  - chaine de caractères saisie par l'utilisateur humain
  - containers de taille variable
    - vecteurs
    - matrices
    - …
  - taille de maillage dépendant de la précision requise
  - fichiers de taille et de structure arbitraire
  - …
- **Solution:** l'allocation dynamique
  - L'allocation dynamique permet d'allouer des *tableaux de taille variable* (tableaux dynamiques)
  - La primitive d'allocation dynamique est *malloc()*
    - déclarée dans le header standard <*malloc.h*>

Exemple

- Allocation dynamique d'un tableau de *K* entiers:
  - *K* étant inconnu à la compilation, on ne peut pas utiliser « *int tab[K];* »

mon_source.cpp

```
#include <malloc.h>
…
int * p = (int*) malloc( K * sizeof(int) ) ;
```



- Accès aux éléments du tableau dynamique:
  - On accède aux membres du tableau par ***arithmétique de pointeurs.***

***(p+2) ⇔ p[2]**

Libération de la mémoire

- La zone-mémoire obtenue par *malloc()* reste allouée tant que le programme ne demande pas explicitement la libération de cette zone-mémoire.
  - **Règle de bonne programmation:** dès qu'une zone-mémoire n'est plus nécessaire, elle doit être libérée par appel à la fonction *free()*.
    - Cf. gestion des ressources

<malloc.h>

```
void  free( void * ptr );
```

- Attention
  - Si *free()* n'est jamais appelé, la mémoire n'est jamais récupérée !
    - C'est ce qui s'appelle une « *fuite-mémoire* » (memory leak)
  - Seul un pointeur obtenu par *malloc()* peut être fourni en argument à *free()*.
    - Sinon, comportement imprévisible (cf. cours gestion-mémoire).

Propriétés de *malloc()*

- *malloc()* renvoie un pointeur sur le début de la zone-mémoire
  - Le premier élément du tableau.

- Avec la version standard de *malloc()*, le programmeur n'a aucun contrôle sur l'emplacement de la zone-mémoire.
  - C'est *malloc()* qui décide tout seul d'après ses structures internes.

- *malloc()* renvoie une zone-mémoire
  - contiguë
  - garantie *au moins aussi grande* que celle qui est demandée
    - parfois ce qui est alloué est légèrement plus grand que demandé.
    - Cela ne change pas grand-chose pour le programmeur…

- *malloc()* renvoie un pointeur *void** (pointeur de type « pointeur sur *void* »)
  - C'est au programmeur de le convertir dans le type désiré (→ via un *cast*)
  - Lorsque *malloc()* échoue, il renvoie un pointeur nul.
    - Par exemple, lorsque la taille demandée dépasse la quantité de mémoire disponible.

Inconvénients de *malloc()*

- *malloc()* est une solution de « bas niveau »
  - C'est le programmeur qui fait le calcul du nombre d'octets dont il a besoin → risque d'erreurs
    - *bug dans l'expression, zéro terminal oublié, …*
  - Ce risque disparaîtra en C++
    - Le C++ propose des primitives de plus haut niveau pour allouer de la mémoire pour « un certain type de variable »

- Risque de fuites-mémoire
  - La zone-mémoire obtenue par *malloc()* reste allouée tant que le programme ne demande pas explicitement la *libération* de cette zone-mémoire.
    - Ceci reste un inconvénient en C++
    - En C++, on peut réduire ce risque en s'appuyant sur des classes qui gèrent correctement leurs allocations-mémoire.

Comparaison entre tableau alloué statiquement / dynamiquement

```
// Tableau alloué statiquement
{
   int i[2] ;
   …
}
```

```
// Tableau alloué dynamiquement
{
   int *p= (int*) malloc( k*sizeof(int) );
   …
}
```

	Tableau alloué statiquement	Tableau alloué dynamiquement
Manipulation de la variable	*Par son nom.*	*Par pointeur.*
Durée de vie	*Limitée au bloc englobant.*	*Jusqu'au free().*
Taille	*La taille doit être connue dès la compilation.*	*La taille peut être connue à l'exécution seulement.*
Où en mémoire ?	*Sur la Pile.*	*Dans le Tas.*

Passage de tableau en argument de fonction

- Deux façons de déclarer la fonction:

```
void fonction( int tab[] ) {
        …
}
```

```
void fonction( int *tab ) {
        …
}
```

- Une seule façon d'appeler:

```
int main() {
        …
        int tab[10] = … ;
        fonction( tab ) ;
        …
}
```

- Attention:
  - si on veut passer explicitement la taille du tableau à la fonction, il faut introduire un paramètre supplémentaire.
    - → *voir exercice sur le decay*

Retour aux pointeurs…