Allocation dynamique de mémoire

Programmation séquentielle en C, 2021-2022

Orestis Malaspinas (A401), ISC, HEPIA

2021-11-09

Inspirés des slides de F. Glück

Allocation dynamique de mémoire (1/8)

- La fonction `malloc` permet d'allouer dynamiquement (pendant l'exécution du programme) une zone de mémoire contiguë.

```c
#include <stdlib.h>
void *malloc(size_t size);
```

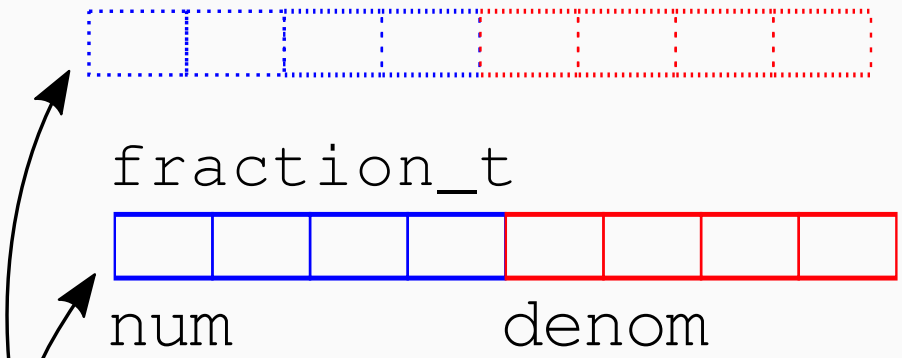
- `size` est la taille de la zone mémoire **en octets**.
- Retourne un pointeur sur la zone mémoire ou `NULL` en cas d'échec: **toujours vérifier** que la valeur retournée est `!= NULL`.

Allocation dynamique de mémoire (2/8)

- On peut allouer et initialiser une `fraction_t`:

```
fraction_t *num = malloc(sizeof(fraction_t));
num->num = 1;
num->denom = -1;
```

- La zone mémoire **n'est pas** initialisée.
- Désallouer la mémoire explicitement ⇒ **fuites mémoires**.




```
fraction_t *frac = malloc(sizeof(fraction_t));
frac = NULL; // addresse mémoire inaccessible
```

Allocation dynamique de mémoire (3/8)

- La fonction `free()` permet de libérer une zone préalablement allouée avec `malloc()`.

```c
#include <stdlib.h>
void free(void *ptr);
```

- Pour chaque `malloc()` doit correspondre exactement un `free()`.
- Si la mémoire n'est pas libérée: **fuite mémoire** (l'ordinateur plante quand il y a plus de mémoire).
- Si la mémoire est **libérée deux fois**: *seg. fault*.
- Pour éviter les mauvaises surprises mettre `ptr` à `NULL` après libération.

Allocation dynamique de mémoire (4/8)

Tableaux dynamiques

- Pour allouer un espace mémoire de 50 entiers: un tableau

```
int *p = malloc(50 * sizeof(int));
for (int i = 0; i < 50; ++i) {
    p[i] = 0;
}
```

Arithmétique de pointeurs

- Parcourir la mémoire différemment qu'avec l'indexation

```
int *p = malloc(50 * sizeof(int));
// initialize somehow
double a = p[7];
double b = *(p + 7); // on avance de 7 "double"
p[0] == *p; // le pointeur est le premier élément
```

Allocation dynamique de mémoire (5/8)

Arithmétique de pointeurs

```
int *p = malloc(16 * sizeof(int))
```

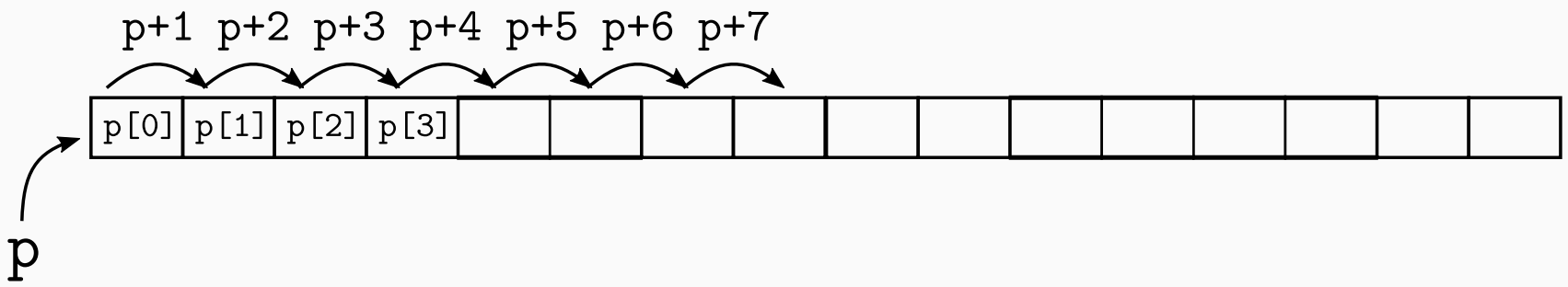


Figure 2: L'arithmétique des pointeurs.

Quelle est la complexité de l'accès à une case d'un tableau?

Allocation dynamique de mémoire (5/8)

Arithmétique de pointeurs

```
int *p = malloc(16 * sizeof(int))
```

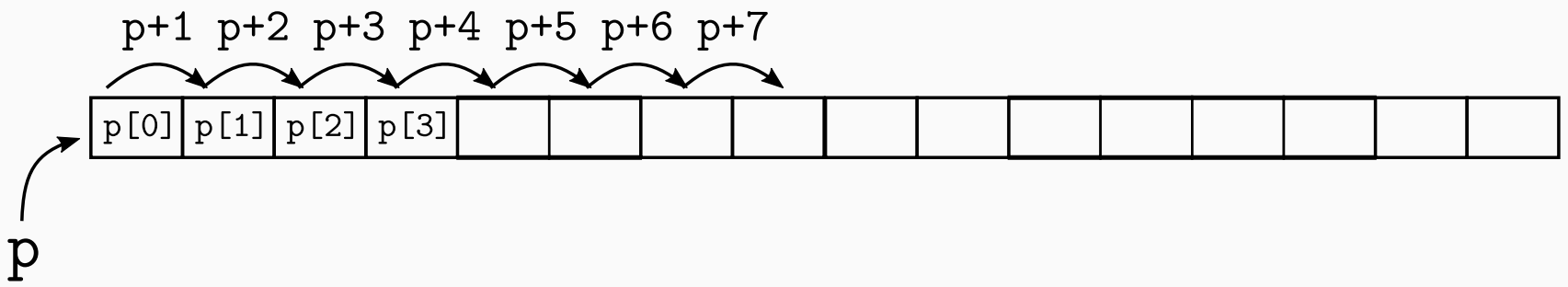


Figure 2: L'arithmétique des pointeurs.

Quelle est la complexité de l'accès à une case d'un tableau?

$\mathcal{O}(1)$.

Allocation dynamique de mémoire (6/8)

Pointeur de pointeur

- Tout comme une valeur a une adresse, un pointeur a lui-même une adresse:

```
int a = 2;
int *b = &a;
int **c = &b;
```

- Chaque * ou & rajoute une indirection.

Allocation dynamique de mémoire (7/8)

Pointeur de pointeur

```
int a = 2;
int *b = &a;
int **c = &b;
...
```

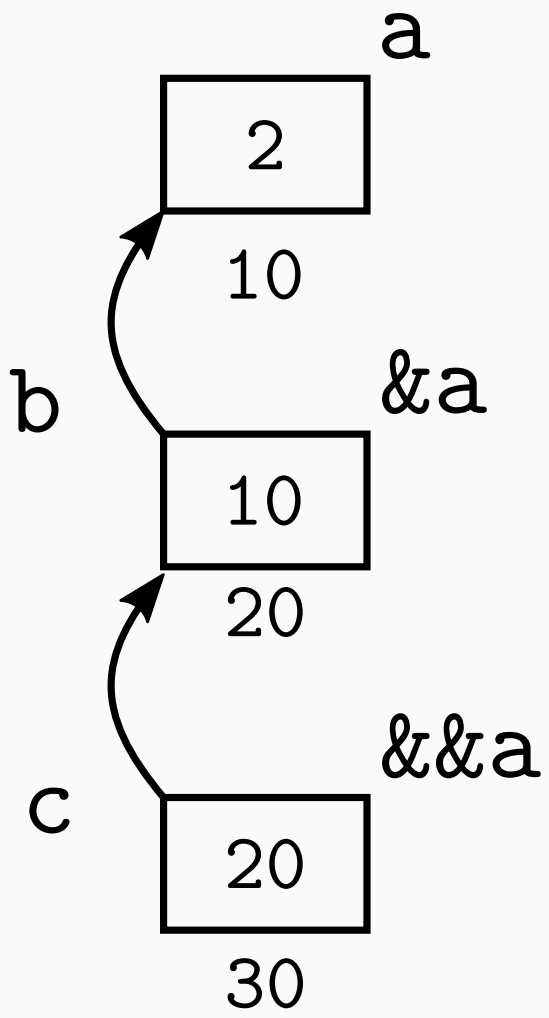


Figure 3: Les références de pointeurs.

Allocation dynamique de mémoire (8/8)

- Avec `malloc()`, on peut allouer dynamiquement des tableaux de pointeurs:

```
int **p = malloc(50 * sizeof(int*));
for (int i = 0; i < 50; ++i) {
    p[i] = malloc(70 * sizeof(int));
}
int a = p[5][8]; // on indexe dans chaque dimension
```

- Ceci est une matrice (un tableau de tableau).

Les *sanitizers*

Problèmes mémoire courants:

- Dépassement de capacité de tableaux.
- Utilisation de mémoire non allouée.
- Fuites mémoire.
- Double libération.

Outils pour leur détection:

- Valgrind (outil externe).
- Sanitizers (ajouts de marqueurs à la compilation).

Ici on utilise les sanitizers (modification de la ligne de compilation):

```
gcc -o main main.c -g -fsanitize=address -fsanitize=leak
```

Attention: Il faut également faire l'édition des liens avec les sanitizers.