Allocation dynamique de mémoire

Programmation séquentielle en C, 2024-2025

Orestis Malaspinas (A401)

2024-12-03

Informatique et Systèmes de Communication, HEPIA

Allocation dynamique de mémoire (1/8)

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

```
#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`.
- Le *type* du retour est `void *` (un pointeur de type quelconque).

Allocation dynamique de mémoire (2/8)

- Allocation *sur le tas* d'un type de base:

```
int *val = malloc(sizeof(int)); // réserve 4 octets sur le tas
*val = 4; // les 4 octets contiennent la valeur 4
```

- Presque la même chose que:

```
int val = 4; // ici l'allocation et l'initialisation
             // sont faites en une fois mais se trouve sur
             // le *tas* (mémoire managée automatiquement)
```

- Attention:

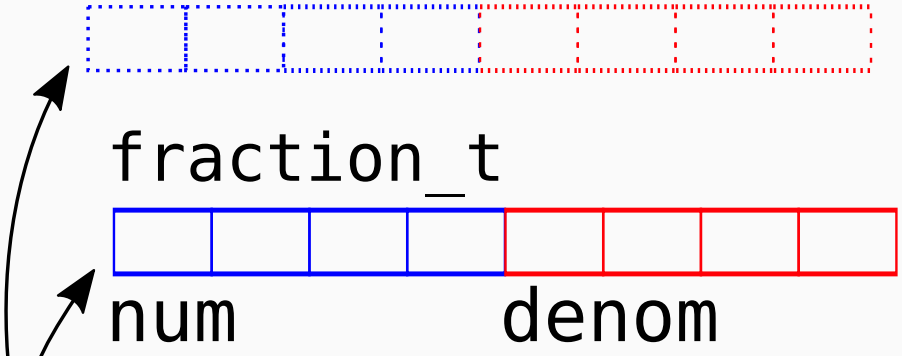
```
int *val = 4; // Ca c'est très faux...
```

Allocation dynamique de mémoire (3/9)

- On peut allouer et initialiser une `fraction_t`:

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

- La zone mémoire **n'est pas** initialisée.
- Désallouer la mémoire explicitement, sinon **fuites mémoires**.
- Il faut connaître la **taille** des données à allouer.




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

Allocation dynamique de mémoire (4/9)

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

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

- A 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 (5/9)

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

Allocation dynamique de mémoire (6/9)

Arithmétique de pointeurs

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

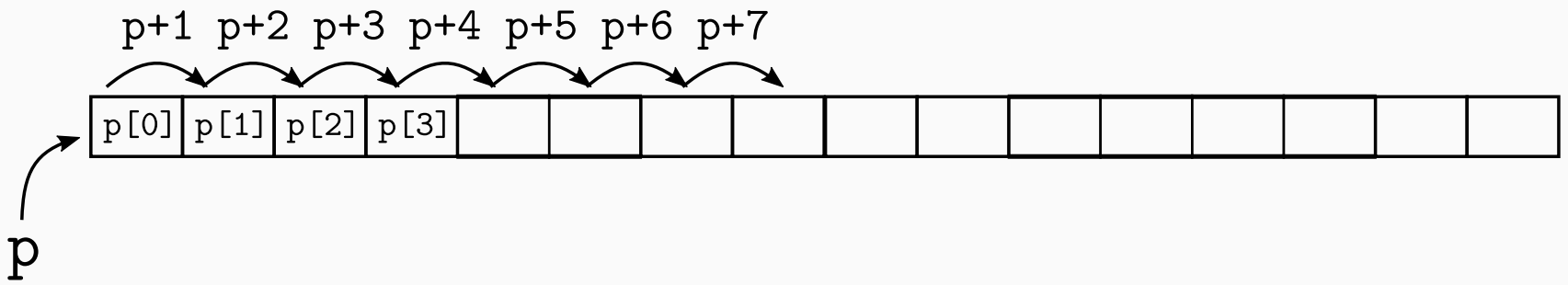


Figure 2: L'arithmétique des pointeurs.

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

Allocation dynamique de mémoire (6/9)

Arithmétique de pointeurs

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

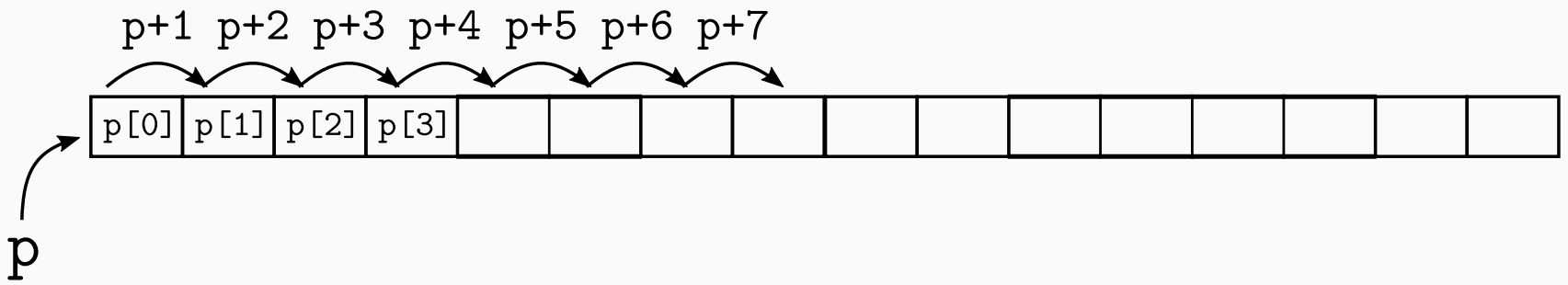


Figure 2: L'arithmétique des pointeurs.

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

$$\mathcal{O}(1).$$

Allocation dynamique de mémoire (7/9)

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;
```

- En effet, un pointeur est aussi une variable (une variable qui contient une adresse mémoire).
- Chaque $*$ rajoute une indirection.

Allocation dynamique de mémoire (8/9)

Pointeur de pointeur

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

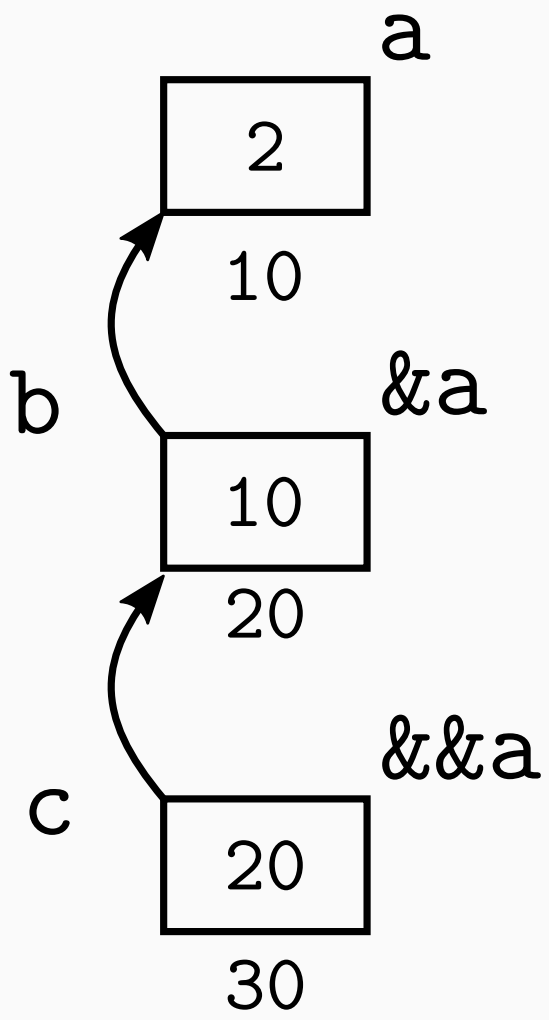


Figure 3: Les références de pointeurs.

Allocation dynamique de mémoire (9/9)

- 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 tableaux).

Tableau dynamique en argument d'une fonction

Implémenter la fonction ci-dessous

```
int32_t *p = malloc(50 * sizeof(*p));
initialize_to(p, 50, -1); // initialise un tableau à -1
free(p); // ne pas oublier
```

Tableau dynamique en argument d'une fonction

Implémenter la fonction ci-dessous

```
int32_t *p = malloc(50 * sizeof(*p));
initialize_to(p, 50, -1); // initialise un tableau à -1
free(p); // ne pas oublier
```

```
void initialize_to(int32_t *p, size_t size, int32_t val) {
    for (size_t i = 0; i < size; ++i) {
        p[i] = val;
    }
}
```

Tableau dynamique retourné d'une fonction

Implémenter la fonction ci-dessous

```
// alloue un tableau de taille 50 et l'initialise à -1
int32_t *p = initialize_to(50, -1);
free(p); // ne pas oublier
```

Tableau dynamique retourné d'une fonction

Implémenter la fonction ci-dessous

```
// alloue un tableau de taille 50 et l'initialise à -1
int32_t *p = initialize_to(50, -1);
free(p); // ne pas oublier
```

```
int32_t *initialize_to(size_t size, int32_t val) {
    int32_t *p = malloc(size * sizeof(*p));
    for (size_t i = 0; i < size; ++i) {
        p[i] = val;
    }
    return p;
}
```

Pourquoi on peut retourner un tableau dynamique et pas un statique?

Tableau dynamique retourné d'une fonction

Implémenter la fonction ci-dessous

```
// alloue un tableau de taille 50 et l'initialise à -1
int32_t *p = initialize_to(50, -1);
free(p); // ne pas oublier
```

```
int32_t *initialize_to(size_t size, int32_t val) {
    int32_t *p = malloc(size * sizeof(*p));
    for (size_t i = 0; i < size; ++i) {
        p[i] = val;
    }
    return p;
}
```

Pourquoi on peut retourner un tableau dynamique et pas un statique?

- Le tableau est alloué sur le **tas** et non sur la **pile**.
- La mémoire est gérée manuellement sur le tas, automatiquement sur la pile.

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, modifiez donc vos *Makefile*):

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

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

Questions

Que fait le code suivant?

```
int *p = malloc(50 * sizeof(int));
p[10] = 1;
```

Questions

Que fait le code suivant?

```
int *p = malloc(50 * sizeof(int));
p[10] = 1;
```

- On alloue de la place pour 50 entiers.
- On initialise le 11ème élément du tableau à 1.
- Les autres éléments sont non-initialisés.

Questions

Que fait le code suivant?

```
float *p = malloc(50);
p[20] = 1.3;
```

Questions

Que fait le code suivant?

```
float *p = malloc(50);
p[20] = 1.3;
```

- On déclare un pointeur de floats de taille 50 octets.
- Mais il ne peut contenir que `50 / 4` floats (un float est composé de 32 bits).
- On dépasse la capacité de la mémoire allouée: comportement indéfini.

Questions

- Soit le code suivant

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

- Le réécrire en utilisant uniquement l'arithmétique de pointeurs.

Questions

- Soit le code suivant

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

- Le réécrire en utilisant uniquement l'arithmétique de pointeurs.

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

Questions

Que valent les expressions suivantes?

```
in32_t *p = malloc(50 * sizeof(int32_t));
sizeof(p);
sizeof(*p);
(p + 20);
*(p + 20);
p[-1];
p[50];
7[p];
```