

Rappels et compléments de C

Cyril Rabat

`cyril.rabat@univ-reims.fr`

Licence 3 Informatique - Info0601 - Systèmes d'exploitation - concepts avancés

2021-2022



Cours n°4 *Rappels de C* *Compléments*

Version 6 janvier 2022

Table des matières

- 1 Écriture et lecture
- 2 Les chaînes de caractères
- 3 Les structures : taille et alignement
- 4 Les structures : allocation et champs dynamiques

Descripteurs de fichier

- Associés à différents types de ressources :
 - ↪ Fichiers
 - ↪ Tubes
 - ↪ Sockets...
- Attention cependant aux propriétés associées au descripteur :
 - ↪ Dépend de l'ouverture (options spécifiées)
 - ↪ Des paramètres par défaut (selon les ressources associées)
- À noter qu'il est possible de modifier les propriétés avec `fcntl`

Écriture et lecture

- Utilisation des appels système `write` et `read`
- Données copiées bit-à-bit :
 - ↪ Pas d'interprétation !
 - ↪ Pointeurs génériques (`void*`)
- Paramètres correspondant aux données à lire/écrire :
 - ↪ Adresse mémoire (pointeur) + taille des données
- Attention cependant à la représentation des données :
 - ↪ Pas de problème entre processus locaux
 - ↪ Problèmes lors de lecture/écriture sur différents hôtes
 - ↪ Problèmes d'architecture, de systèmes, etc.

Utilisation des pointeurs génériques

- Peuvent représenter tout type de donnée (ou structure)
- Pour accéder aux données, transtypage (*cast*)
- Attention à l'ordre :
 - Soit : `void *ptr`
 - `(int) (*ptr)` → **Interdit !**
 ↪ dereferencing 'void *' pointer
 - `*(int*) ptr` → **Autorisé**

Taille des données : `sizeof`

- `sizeof` permet de retourner la taille des données en octets
- Important : ne pas spécifier la taille directement dans le code !
 - ↪ Portable (suivant l'architecture, le système...)
 - ↪ Évite les erreurs!

Exemples

- Avec `int i` :
 - ↪ `sizeof(i) = int = 4o`
- Avec `int t[10]` :
 - ↪ `sizeof(t) = int[10] = 10×4o`

Cas des pointeurs

- Tailles différentes en 32 bits (4o) et 64 bits (8o)
- Éviter les confusions entre pointeur et données pointées

Exemples

- Avec `int *i` :
 - ↪ `sizeof(i) = int* = 8o` (sur 64 bits)
 - ↪ Taille indépendante de l'initialisation de `i`
- Attention à la taille de ce qui est pointé :
 - ↪ `sizeof(*i) = 4o`

Résumé

- Transtypage du pointeur générique **avant** de récupérer sa valeur pointée
- Pas d'interprétation lors de la lecture/écriture avec `read/write`
↪ Copie bit à bit
- Pour une variable `x` de type primitif `type`
↪ `sizeof(x) = sizeof(type)`
- Taille d'un tableau statique de `n` cases : `sizeof(type) * n`
- Taille d'un pointeur :
 - Indépendant du type
 - Indépendant de l'initialisation
- **La taille est déterminée à la compilation**

Pour la taille des données, utilisez `sizeof`

Les chaînes de caractères en C

- Source de nombreux *segmentation fault* :
 ↪ Accès mémoires interdits
- Erreurs courantes :
 - Espace mémoire non alloué (avec `char*`)
 - Dépassement de la capacité allouée
 - Mauvaise maîtrise des fonctions de la bibliothèque (`string.h`)
 - Problème du caractère de fin `'\0'`
 - Confusions `char*` et `char[]`

Utilisation de `scanf` pour les chaînes de caractères

- `scanf` avec `%s` :
 - ↪ Spécification obligatoire de la longueur maximale
- Exemple : `scanf("%10s", s)`
 - ↪ **Attention** : la chaîne doit être allouée et de taille 11 (pour le `'\0'`)
- Pour la gestion des espaces :
 - `scanf("%10[A-Z]", s)` :
 - ↪ 10 lettres majuscules maximum + espace
 - `scanf("%10[^\n]", s)` :
 - ↪ Tout sauf le retour à la ligne

Comment lire plusieurs chaînes ?

- Avec `%s`, lecture jusqu'au délimiteur
- Ce dernier reste dans le tampon !
- Rappel : ne pas utiliser `fflush` !
 ↪ Possible uniquement sur les flux en sortie
- Solution : lire tous les caractères restants un par un

Vider le tampon d'entrée

```
char c;
```

```
while(((c = getchar()) != '\n') || (c == EOF));
```

Exemple de code complet

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char s1[10];
    char s2[10];
    char c;

    printf("Saisir_votre_nom:_");
    if (scanf("%9s", s1) != 1) {
        fprintf(stderr, "Erreur_du_scanf\n"); exit(EXIT_FAILURE);
    }
    while (((c = getchar()) != '\n') || (c == EOF));

    printf("Saisir_votre_prenom:_");
    if (scanf("%9s", s2) != 1) {
        fprintf(stderr, "Erreur_du_scanf\n"); exit(EXIT_FAILURE);
    }
    while (((c = getchar()) != '\n') || (c == EOF));

    return EXIT_SUCCESS;
}
```

Utilisation de `gets` et `fgets`

- `gets` et `fgets` permettent de lire des chaînes de caractères :
 - ↪ Utilisation du délimiteur retour à la ligne ou EOF
 - ↪ Simplifie la lecture
- Mais ne pas utiliser `gets` : fonction dépréciée !
 - ↪ Impossible de fixer une taille maximale !
- Sur l'utilisation de `fgets` :
 - La taille du buffer est spécifiée
 - Contrairement à `scanf` : le `'\0'` est compris dans la taille
 - Le délimiteur est lu et stocké dans la chaîne

Premier exemple

```
int main() {  
    char buffer1[16] = "Bonjour";  
    char buffer2[16] = "Au_revoir";  
  
    printf("Chaines_:_%s_et_%s\n", buffer1, buffer2);  
  
    return EXIT_SUCCESS;  
}
```

- Code correct :
 ↪ '\0' ajouté à la compilation

Deuxième exemple

```
int main() {  
    char buffer1[16];  
    buffer1 = "Bonjour";  
    char buffer2[16];  
    buffer2 = "Au_revoir";  
  
    printf("Chaines_:_%s_et_%s\n", buffer1, buffer2);  
  
    return EXIT_SUCCESS;  
}
```

- Code incorrect :

↪ Erreur de compilation : types incompatibles char* et char[]

Troisième exemple

```
int main() {  
    char buffer1[16];  
    buffer1 = "Bonjour";  
    char buffer2[16];  
    buffer2 = "Au_revoir";  
  
    free(buffer1);  
    free(buffer2);  
  
    printf("Chaines_: _%s_et_%s\n", buffer1, buffer2);  
  
    return EXIT_SUCCESS;  
}
```

- Code incorrect :

↪ Erreur de compilation : tentative de libérer un objet qui n'est pas dans le tas

Quatrième exemple

```
int main() {  
    char buffer1[16] = "abcdefghijklmnop";  
    char buffer2[16] = "abcdefghijklmnop";  
  
    printf("Chaines_:_%s_et_%s\n", buffer1, buffer2);  
  
    return EXIT_SUCCESS;  
}
```

- Affichage :

↪ Chaines : abcdefghijklmnopabcdefghijklmnop et abcdef...

- '\0' non ajouté par manque de place !

Stockage de chaînes dans un fichier

- Exemple : `char str[10] = "toto\0";`
- ❶ Doit-on stocker tous les caractères alloués ?
 - ↪ Utilisation de `sizeof(char) × 10`
 - ↪ Stocké : `toto ?????` (`10 × sizeof(char)` octets)
- ❷ Uniquement les caractères utiles (avant le `'\0'`) ?
 - ↪ Utilisation de `strlen(str)`
 - ↪ Stocké : `toto` (`4 × sizeof(char)` octets)
- ❸ Les caractères + le `\0` ?
 - ↪ Utilisation de `strlen(str) + 1`
 - ↪ Stocké : `toto` (`5 × sizeof(char)` octets)

Exemples de codes (sans gestion d'erreur)

```
/* Écriture */
char str[10] = "Cool";
int fd;

fd = open("toto.bin", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);

write(fd, str, sizeof(char) * 10);

close(fd);

/* Lecture */
char str[10];
int fd;

fd = open("toto.bin", O_RDONLY, S_IRUSR|S_IWUSR);

read(fd, str, sizeof(char) * 10);
printf("Lu: '%s'\n", str);

close(fd);
```

Comment lire une chaîne de taille variable ?

- ❶ Si la taille est connue :
 ⇨ Pas de problème !
- ❷ Si la taille est inconnue :
 ⇨ Lecture caractère par caractère jusqu'à '`\0`'...
 ⇨ ... À condition que '`\0`' soit présent !

La lecture caractère par caractère est à éviter car plus longue !
Il y a de multiples appels systèmes

Lecture caractère par caractère (sans gestion d'erreur)

```
/* Écriture */
char str[10] = "Cool";
int fd;

fd = open("toto.bin", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
write(fd, str, sizeof(char) * (strlen(str) + 1));

close(fd);

/* Lecture */
char str[10];
int fd, i;

fd = open("toto.bin", O_RDONLY, S_IRUSR|S_IWUSR);

i = 0;
while((read(fd, &str[i], sizeof(char)) == sizeof(char)) && (str[i] !=
    '\0'))
    i++;
printf("Lu:_%s'\n", str);

close(fd);
```

Autre solution : ajout de la taille (sans gestion d'erreur)

```
/* Écriture */
char str[10] = "Cool";
int fd, taille;

fd = open("toto.bin", O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
taille = strlen(str) + 1;
write(fd, &taille, sizeof(int));
write(fd, str, sizeof(char) * taille);
close(fd);

/* Lecture */
char *str;
int fd, taille;

fd = open("toto.bin", O_RDONLY, S_IRUSR|S_IWUSR);
read(fd, &taille, sizeof(int));
str = malloc(sizeof(char) * taille);
read(fd, str, sizeof(char) * taille);
printf("Lu:_%s'\n", str);
free(str);
close(fd);
```

Résumé sur les chaînes de caractères

- Possibilité d'initialiser avec une chaîne statique :
 - `char *toto = "Bonjour";`
 - `char toto[10] = "Bonjour";`
- Attention au `'\0'` :
 - Il est ajouté automatiquement sur les chaînes statiques
 - La taille déclarée doit en tenir compte
 - Pris en compte ou non suivant des fonctions utilisées
- L'affectation d'une chaîne statique n'est possible qu'avec les `char*`
↳ Sinon, utilisation de `strcpy` ou de `snprintf`
- Attention à l'allocation : `char[]` vs `char*`
↳ N'oubliez pas le `free`
- Interdiction d'utiliser `scanf` sans limitation de taille et `gets`
- Stockage dans un fichier :
 - ↳ Soit stocker le `'\0'` pour lecture caractère par caractère (à éviter)
 - ↳ Soit stocker la taille + les caractères

Exemple (1/2)

```
typedef struct {
    char nom[256];
    char prenom[256];
    int age;
} personne_t;

void methode(personne_t p) {
    printf("methode_1_:_%s_%s_(%d_an(s))\n", p.nom, p.prenom, p.age);
    p.nom[2] = '\0'; p.age = 30;
    printf("methode_2_:_%s_%s_(%d_an(s))\n", p.nom, p.prenom, p.age);
}

int main() {
    personne_t p1 = { "Toto", "Tata", 40 };

    methode(p1);
    printf("main_:_%s_%s_(%d_an(s))\n", p1.nom, p1.prenom, p1.age);

    return EXIT_SUCCESS;
}
```


Exemple (2/2)

Affichage obtenu

```
methode 1 : Toto tata (40 an(s))  
methode 2 : To tata (30 an(s))  
main : Toto tata (40 an(s))
```

Remarques

- Paramètre : passage par copie
- Données entièrement recopiées :
 ↪ Même chose avec une simple affectation

Avec des pointeurs (1/2)

```
typedef struct {
    char *nom;
    char *prenom;
    int age;
} personne_t;

void methode(personne_t p) {
    printf("methode_1:_%s_%s_(%d_an(s))\n", p.nom,p.prenom,p.age);
    p.nom[2] = '\\0';p.age = 30;
    printf("methode_2:_%s_%s_(%d_an(s))\n", p.nom,p.prenom,p.age);
}

int main() {
    personne_t p1;
    p1.nom = malloc(sizeof(char) * 5);
    p1.prenom = malloc(sizeof(char) * 5);
    strcpy(p1.nom, "Toto");strcpy(p1.prenom, "Tata");p1.age = 40;
    methode(p1);
    printf("main:_%s_%s_(%d_an(s))\n", p1.nom, p1.prenom, p1.age);
    free(p1.nom);free(p1.prenom);
    return EXIT_SUCCESS;
}
```

Avec des pointeurs (2/2)

Affichage obtenu

methode 1: Toto tata (40 an(s))

methode 2: To tata (30 an(s))

main: To tata (40 an(s))

Remarques

- Données entièrement recopiées :
 - Pour l'entier : OK
 - Pour les pointeurs : adresses recopiées
 - Données pointées non recopiées

Taille des structures

```
typedef struct {  
    char nom[256];  
    char prenom[256];  
    int age;  
} personne_t;
```

```
personne_t p;  
personne_t *ptr;
```

Description

- `sizeof(p) = sizeof(personne_t) = 516o`
- `sizeof(ptr) = 8o` (ptr initialisé ou non)
- `sizeof(*ptr) = 516o` (ptr initialisé ou non)

Écriture/lecture

```
typedef struct {
    char nom[256];
    char prenom[256];
    int age;
} personne_t;

int main() {
    personne_t p = {"Smith", "John", 30}, p2;
    int fd;

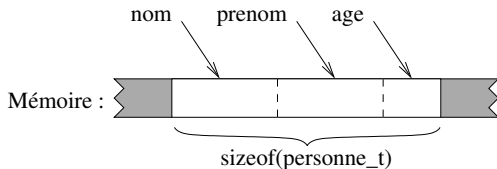
    fd = open("toto.bin", O_WRONLY | O_CREAT | O_TRUNC,
              S_IRUSR | S_IWUSR);
    write(fd, &p, sizeof(personne_t));
    close(fd);
    fd = open("toto.bin", O_RDONLY, S_IRUSR | S_IWUSR);
    read(fd, &p2, sizeof(personne_t));
    printf("%s_ %s_ (%d_ ans)\n", p2.prenom, p2.nom, p2.age);
    close(fd);

    return EXIT_SUCCESS;
}
```

Gestion mémoire

- Données des structures : stockées de manière contigüe
- Avantages :
 - Manipulation aisée de jeux de données
 - Lecture/écriture bit-à-bit possible
- Attention à la taille de la structure !

```
typedef struct {  
    char nom[256];  
    char prenom[256];  
    int age;  
} personne_t;
```



Alignement mémoire (1/2)

- Accès mémoire par le CPU :
 - Adresses mémoire multiples de mot (`word`)
 - Un mot = 4 octets en 32 bits et 8 octets en 64 bits
- Pour améliorer les performances :
 - Accès immédiat aux données (et non en plusieurs accès)
 - Alignement des données
 - Ajout d'octets de bourrage
- Lors de la compilation :
 - Analyse de la structure + recherche du plus grand champ
 - Alignement de toute la structure en fonction du plus grand champ
 - Alignement propre de chaque champ
 - Ajout d'octets de bourrage entre les champs si nécessaire

Alignement mémoire (2/2)

Tailles et alignement des représentations - Linux et gcc

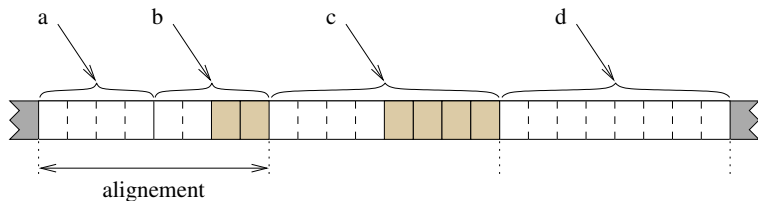
Type	Taille	Aligne.	Type	Taille	Aligne.
char	1o	1o	short	2o	2o
int	4o	4o	long	4o / 8o	4o / 8o
float	4o	4o	double	8o	4o / 8o
long long	8o	8o	long double	12o / 16o	4o / 16o
pointer	4o / 8o	4o / 8o			

Légende : Xo = 32 et 64 bits, Xo 32 bits, Xo 64 bits

- L'alignement global dépend du plus grand champ :
 ↪ Exemple : si short alignement sur 2o, si int alignement sur 4o
- Il dépend aussi du système et du compilateur :
 ↪ gcc, Visual C++, C++ builder

Représentation mémoire - exemples (1/2)

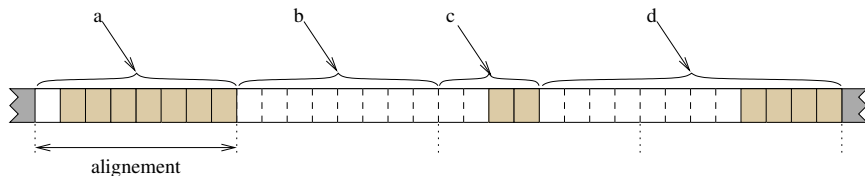
```
typedef struct {  
    unsigned int a;  
    short b;  
    float c;  
    double d;  
} structure1_t;
```



Représentation mémoire

Représentation mémoire - exemples (2/2)

```
typedef struct {  
    char a;  
    double b;  
    char c[2];  
    int d[2];  
} structure2_t;
```



Représentation mémoire

Corriger les alignements (1/2)

- Automatique avec les compilateurs
- Pour gcc, possibilité d'ajouter l'option `-Wpadded` :
 ↪ Affiche des avertissements en cas de mauvais alignements
- Comment aligner manuellement les champs ?
 - Réorganisation des champs
 - Ajout de champs de bourrage : `char _pad1[X]`

```
typedef struct {
    unsigned int a;
    short b;
    float c;
    double d;
} structure1_t;
```

Non alignée

```
typedef struct {
    unsigned int a;
    short b;
    char _pad1[2];
    float c;
    char _pad2[4];
    double d;
} structure1b_t;
```

Alignée



Corriger les alignements (2/2)

- Automatique avec les compilateurs
- Pour gcc, possibilité d'ajouter l'option `-Wpadded` :
 ↪ Affiche des avertissements en cas de mauvais alignements
- Comment aligner manuellement les champs ?
 - Réorganisation des champs
 - Ajout de champs de bourrage : `char _pad1[X]`

```
typedef struct {
    char a;
    double b;
    char c[2];
    int d[2];
} structure2_t;
```

Non alignée (32o)

```
typedef struct {
    char a;
    char c[2];
    char _pad1[5];
    double b;
    int d[2];
} structure2b_t;
```

Alignée (24o)

La correction manuelle n'est pas à faire... sauf cas très exceptionnels!

Résumé sur la taille et l'alignement des structures

- Alignement des champs en fonction :
↔ Du type des champs, du “plus grand type” de champ, du compilateur
- Alignement "manuel" non nécessaire MAIS :
 - Code possiblement non portable si accès bit-à-bit
 - Accès possible sur un champ donné dans un fichier
- Rappel de l'algorithme général pour l'alignement :
 - ➊ Recherche du plus grand champ : alignement global de la structure
 - ➋ Alignement de chaque champ en fonction du type
 - ➌ Ajout d'octets de bourrage entre les champs si nécessaire
 - ➍ Ajout d'octets de bourrage à la fin la structure

N'utilisez pas l'option `-Wpadded` dans vos projets.

Allocation dynamique d'une structure (1/2)

```
/* Première structure */
typedef struct {
    char nom[256];
    char prenom[256];
    int age;
} personneS_t;

/* Allocation dynamique (sans gestion d'erreur) */
personneS_t *p;
if((p = malloc(sizeof(personneS_t))) == NULL) {
    perror("Erreur_lors_de_l'allocation_");
    exit(EXIT_FAILURE);
}

/* Libération mémoire */
free(p);
```

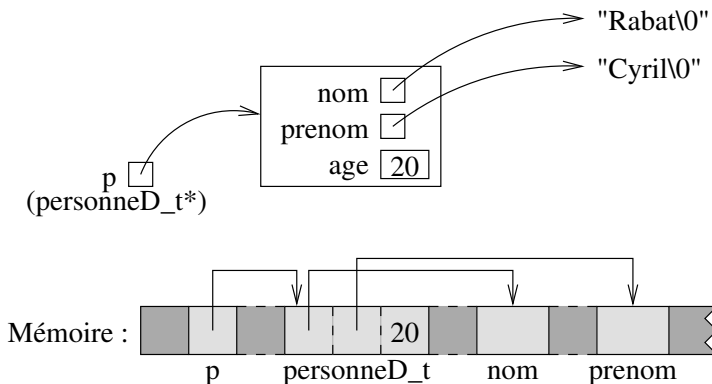
Allocation dynamique d'une structure (2/2)

```
/* Deuxième structure */
typedef struct {
    char *nom;
    char *prenom;
    int age;
} personneD_t;

/* Allocation dynamique (sans gestion d'erreur) */
personneD_t *p;
p = malloc(sizeof(personneD_t));
p->nom = malloc(sizeof(char) * 256);
p->prenom = malloc(sizeof(char) * 256);

/* Libération mémoire */
free(p->nom);
free(p->prenom);
free(p);
```

Représentation mémoire



Écriture dans un fichier

```
/* Attention : pas de gestion d'erreur ici ! */

fd = open("toto.bin", O_CREAT | O_TRUNC | O_WRONLY,
          S_IRUSR | S_IWUSR);

taille = strlen(p->nom) + 1;
write(fd, &taille, sizeof(int));
write(fd, p->nom, taille * sizeof(char));

taille = strlen(p->prenom) + 1;
write(fd, &taille, sizeof(int));
write(fd, p->prenom, taille * sizeof(char));

write(fd, &(p->age), sizeof(int));
close(fd);
```

Question rapidité (1/2)

Code 1

```
void methode(personne_t p) { }

int main() {
    int i;
    personne_t p1;

    strcpy(p1.nom, "Toto");
    strcpy(p1.prenom, "Tata");
    p1.age = 40;

    for(i = 0; i < 100000000; i++)
        methode(p1);

    return EXIT_SUCCESS;
}
```

Code 2

```
void methode(personne_t *p) { }

int main() {
    int i;
    personne_t p1;

    strcpy(p1.nom, "Toto");
    strcpy(p1.prenom, "Tata");
    p1.age = 40;

    for(i = 0; i < 100000000; i++)
        methode(&p1);

    return EXIT_SUCCESS;
}
```

Question rapidité (2/2)

- Le code 2 produit le programme le plus rapide
- Passage par valeur : copie de tous les champs
- Passage par adresse : uniquement l'adresse
- Sans l'option `-O3`, facteur 30 (suivant configuration)

Autre question sur la rapidité (1/2)

```
/* Boucle 1 */
for(i = 0; i < 100000000; i++) {
    personne_t p1;
    strcpy(p1.nom, "Toto");
    strcpy(p1.prenom, "Tata");
    p1.age = 40;
}
/* Boucle 2 */
for(i = 0; i < 100000000; i++) {
    personne_t *p1 = malloc(sizeof(personne_t));
    strcpy(p1->nom, "Toto");
    strcpy(p1->prenom, "Tata");
    p1->age = 40;
    free(p1);
}
/* Boucle 3 */
for(i = 0; i < 100000000; i++) {
    personne_t *p1 = malloc(sizeof(personne_t));
    strcpy(p1->nom, "Toto");
    strcpy(p1->prenom, "Tata");
    p1->age = 40;
}
```

Autre question sur la rapidité (2/2)

- Boucle 1 plus rapide :

```
for(i = 0; i < 100000000; i++) {  
    personne_t p1;  
    strcpy(p1.nom, "Toto");  
    strcpy(p1.prenom, "Tata");  
    p1.age = 40;  
}
```

- Facteur 10 entre les deux premières boucles
 ↪ Allocation dynamique coûteuse!
- Pour la troisième (sans libération de la mémoire) :
 - Dépend de la configuration (mémoire)
 - Peut être plus rapide jusqu'à un certain nombre de tours
 ↪ Sauf s'il est tué avant...
 - Mais de toutes façons : **c'est à proscrire!**

Résumé sur l'allocation dynamique des structures

- Si la structure ne possède pas de champ "dynamique" :
 - ↪ Allocation directe (utilisation de `sizeof`)
- Sinon :
 - ↪ Allocation de la structure
 - ↪ Allocation de chaque champ dynamique
- Écriture/lecture :
 - ↪ En une fois si tous les champs sont statiques
 - ↪ Champ par champ sinon
- Libération mémoire :
 - ↪ Si tous les champs sont statiques : un seul appel à `free`
 - ↪ Sinon :
 - `free` pour chaque champ dynamique
 - Puis libération mémoire correspondant à la structure
- L'allocation dynamique doit être utilisée uniquement si nécessaire