C : Bases 3 : pointeurs, stockage, allocation

Pierre Karpman

Lycée Champollion MP2I

https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Roadmap

Poly, Sections 5 & 6

Table des matières

Type pointeur

Adresses

En C, la plupart des objets (mais pas tous !) *peuvent* se voir attribuer une adresse mémoire

- Plus de détails sur l'organisation de cette mémoire dans un prochain cours

Définition

Une variable de type pointeur stocke la valeur de l'adresse mémoire d'un objet

La manipulation directe d'adresses mémoire *via* des pointeurs est source fréquente de bugs ; utilisez systématiquement l'*address sanitizer* dans ces cas

Syntaxe de déclaration

Le type « pointeur vers `t` » s'écrit « `t*` » ou « `t *` » :

```
t *p; // variable p de type « pointeur vers le type t »
t* q; // — q —
```

Type pointeur *bis*

Exemples

```
int *p1; // pointeur vers un int
int **p2; // pointeur vers un (pointeur vers un int)
struct gcof *p3; // pointeur vers une struct gcof
```

Pointeur nul

N'importe quelle variable (ou expression) de type pointeur peut prendre la valeur « nulle », qui désigne l'*absence* d'une adresse ; plusieurs syntaxes :

```
int *p = NULL; // classique
int *q = nullptr; // moderne
```

Pointeur vers `void`

Il existe un type particulier de *pointeur vers le type void* : `void *`. N'importe quel type pointeur peut être converti vers `void *` et vice-versa, mais on ne peut pas faire grand chose avec un `void *`

Affichage & conversion

Conversion `t*` $\leftrightarrow$ `void *`

On peut généralement tenter de convertir explicitement le type d'une expression avec la syntaxe :

```
(nouveau type)expression
```

Pour `void *` :

```
int *p = NULL;
void *q = (void *)p;
int *r = (int *)q;
```

Affichage

On peut afficher la « valeur » d'un pointeur `void *` avec la spécification de conversion `%p`

Démo.

Déréférencement

Principe

On peut déréférencer un pointeur pour accéder (lire, écrire) à la zone mémoire dont l'*adresse* est donnée par la valeur du pointeur

Syntaxe

Le déréférencement se fait *via* l'opérateur préfixe unaire de déréférencement « * » :

```
*p
```

Exemples

```
int *p;
int a;
// ...
*p = 12; // on écrit 12 dans   la zone mémoire pointée par p
a = *p; // on lit la valeur de la —
p = 13; // erreur de type
```

Erreurs au déréférencement

Règle

Pour qu'un déréférencement soit valide, il faut que l'adresse contenue dans le pointeur soit initialisée et valide… (Par exemple : pas nulle)

Démo.

Obtention d'adresse

Principe

On peut obtenir l'adresse d'un objet qui en possède une (typiquement une variable, mais pas un littéral) *via* l'opérateur « *address of* » `&`

Syntaxe

`&x`

Exemples

```
int a;
int *p = &a; // pointeur initialisé avec l'adresse de la variable a
int t[12] = {};
int *q = &t[3]; // pointeur initialisé avec l'@ du 4ième élément de t
```

`*`, `&` : typage, relation

- Si `e` est de type `t*`, alors `*e` est de type `t`
- Si `e` est de type `t`, alors `&e` est de type `t*`
- `&(*e)` et `*(&e)` sont équivalentes à `e` *si valides*

Passage d'arguments de type pointeur

- On écrit les types comme d'habitude, et le passage se fait par valeur comme d'habitude
- Mais les valeurs *sont des adresses* ce qui permet d'émuler un passage par *référence*

Exercice

1. Que fait la fonction suivante ?

```
void myst(int *x, int *y)
{
    int t = *x;
    *x = *y;
    *y = t;
}
```

2. Comment l'invoquer ?

Déréférencement « à la tableaux »

Problème

Il se peut (c'est courant) qu'un pointeur `p` de type `t*` ait pour valeur le « début » d'une zone mémoire pour stocker $n > 1$ objets *de même type* `t`

- `*p` permet d'accéder au *premier* ou zéroième élément
- *quid* des autres ?

Une solution

On peut utiliser une syntaxe « à la tableaux » :

- `p[0]` est équivalent à `*p`
  - À éviter si `p` ne pointe que vers un seul élément
- `p[1]` accède au second ou unième élément
- *etc.*

Caveat

Pour qu'un déréférencement soit valide, il faut (comme pour un tableau) que l'adresse le soit

Tableaux $\rightsquigarrow$ pointeurs

Principe

Un argument de fonction de type tableau se *dégrade* en une variable de type pointeur à l'intérieur d'une fonction (`t[...]` $\rightsquigarrow$ `t*`).
Ceci explique que la modification d'un tableau à l'intérieur d'une fonction persiste au delà de son exécution

Démo.

Pointeurs de `struct`

Il est assez courant de manipuler des enregistrements *via* des pointeurs ; le langage propose un raccourci syntaxique rendant cette manipulation plus aisée :

```
s->a // équivalent à (*s).a
```

Utilité des pointeurs

Principalement :

- Émuler des passages par référence (*cf.* ci-dessus)
- Référencer des zones mémoires allouées « dynamiquement » (*cf.* ci-dessous)
- Construire des types récursifs (*cf.* plus tard)

Table des matières

Durée de vie

Principe

En C, les objets ont une *durée de vie* qui détermine quand il est possible (légalement) d'y accéder *au cours de l'exécution du programme*
La durée de vie d'un objet est déterminée par sa *durée de stockage*

Durée de vie $\neq$ portée

- Ce n'est pas parce qu'on peut faire référence à une variable que l'objet qu'elle désigne est vivant
  - Bugs typiques : *use after free*, *use after return*, *use after scope* (le nom est mal choisi…)
- Ce n'est pas parce qu'on ne peut plus référencer un objet qu'il est mort
  - Bug typique : *fuites mémoires*

Remarque

Certains langages (mais pas le C, et c'est rare) permettent de *réifier* la durée de vie des objets afin d'éviter les bugs liés à celle-ci

Types de durée de stockage

Statique

Durée de stockage correspondant à toute l'exécution du programme. Typiquement pour les variables globales.

Automatique

Durée de stockage typique (par défaut) des variables-paramètres & variables locales ; correspond à la durée *d'exécution* du bloc où la variable est introduite.

- Attention : un appel de fonction dans un bloc ne termine pas l'exécution de celui-ci !
- Seuls les `return` ou le fait d'atteindre la fin du bloc mettent fin à celle-ci

Allouée

Durée de stockage des objets alloués dynamiquement (*cf.* ci-dessous). Commence à l'allocation et termine à la libération. Adapté aux objets dont la durée doit être comprise entre automatique et statique, ou de taille inconnue à la compilation.

Impact de la durée de stockage : un bug

Démo.

Table des matières

Allocation dynamique de mémoire

Principe

L'allocation dynamique de mémoire permet de créer des objets de durée de stockage allouée. Leur durée de vie est contrôlée explicitement par la ou le programmeur, ce qui est une source interminable de bugs.

Usages typiques

- Créer des objets de durée de vie plus longue qu'avec un stockage automatique, qu'on ne peut/veut pas allouer avec un stockage statique.
- Créer de gros objets (pour des raisons concrètes indépendantes du langage, *cf.* plus tard)

Discipline

Tout programme effectuant des allocations dynamiques doit être compilé avec `-fsanitize=address` ; les responsabilités d'allocation & de libération doivent être clairement spécifiées

Mécanisme

Allocation & libération

La création d'un objet de durée de vie allouée se fait par appel de la fonction `malloc` ou similaire ; la libération (fin de la durée de vie) par appel de la fonction `free` ou similaire

`malloc` (dans `stdlib.h`)

Signature : `void *malloc(size_t size)`
Spécifications : alloue `size` octets de mémoire non initialisée de durée de stockage allouée, et renvoie un pointeur contenant l'adresse du début de la zone (ou `NULL` si l'allocation a échoué)

`free` (dans `stdlib.h`)

Signature : `void free(void *p)`
Spécifications : si `p` est nul ou de valeur égale à une adresse renvoyée par `malloc` ou similaire *qui n'a pas encore été libérée*, alors libère (met fin à la durée de vie) de l'objet correspondant ; dans tous les autres cas : UB !

`malloc` : compléments

Conversion de type

La fonction `malloc` renvoie un pointeur de type `void *`, qui en pratique sera convertit vers un pointeur vers un autre type. Il n'est *pas nécessaire* d'effectuer *explicitement* cette conversion

Calcul de la taille : `sizeof`

L'argument de `malloc` est une taille en octets, qui doit correspondre à celle de l'objet que l'on veut utiliser

- Mais quelle est la taille d'un `int` ?
- D'un type enregistrement défini par nous même ?

La taille d'un objet de type `t` peut s'obtenir avec l'opérateur `sizeof` comme `sizeof(t)`

Exemples

```
int *p = malloc(sizeof(int)); // allocation pour stocker un int
int *q = malloc(10 * sizeof(int)); // — dix —
struct gcof *r = malloc(sizeof(struct gcof)); // — une struct gcof
```

`malloc` : gestion d'erreur

`malloc` peut renvoyer `NULL` (typiquement si l'on demande *vraiment* trop de mémoire, *cf.* prochain cours)

- Il faut donc en principe vérifier sa valeur de retour pour gérer une éventuelle erreur
- Cette vérification ne doit *pas* se faire *via* `assert` :

```c
int *p = malloc(1000000000 * sizeof(int));
assert(p != NULL); // non
```

  - `assert` : vérification de non respect des préconditions, d'invariants faux… : problème d'usage et de logique, *pas* problème de ressources (ou d'accès, etc.)

```c
int *p = malloc(1000000000 * sizeof(int));
if (p == NULL) { // idéalement
   fprintf(stderr, "%s: malloc failure\n", __LINE__);
   exit(EXIT_FAILURE);
}
```

`free` : compléments

- Appeler `free` avec un argument nul ne fait rien (c'est légal)
- Appeler `free` avec un argument « déjà libéré » est UB
- *Lire* la valeur d'un pointeur donné comme argument à `free` est UB !!

Fuites mémoire

Définition

Une *fuite mémoire* désigne la présence d'objets de durée allouée « encore vivants » qu'on ne peut plus référencer : on ne peut plus utiliser l'objet *ni libérer la mémoire qu'il occupe*

Fuites mémoire : erreur

Les fuites mémoire sont des problèmes qu'il faut corriger quand ils apparaissent

Les programmes C écrits en CPGE doivent (sauf mention expresse du contraire) être **sans fuite mémoire**

Fuites mémoire : exemple

```c
#include <stdlib.h>

void fun(void) {
    int *p = malloc(sizeof(int));
}

int main(void) {
    fun();
    int *p = malloc(sizeof(int));
    p = NULL;

    return EXIT_SUCCESS;
}
```

La compilation avec l'*address sanitizer* permet de détecter beaucoup de fuites mémoire

Fuites mémoire : exemple 2

```c
#include <stdlib.h>

struct dbint
{
    int *p;
};

int main(void) {
    struct dbint *dbi = malloc(sizeof(struct dbint));
    dbi->p = malloc(sizeof(int));
    *(dbi->p) = 12;
    free(dbi);

    return EXIT_SUCCESS;
}
```

Fuites mémoire : prévention

- Définir une politique d'allocation & libération rigoureuse (qui fait quoi, quand)
- Écrire des fonctions dédiées pour les allocations & libérations complexes
- Utiliser des tests ; tester son programme avec l'*address sanitizer* (ou *valgrind*)

Bestiaire d'erreurs

Démo.