

L3 MI — Programmation

Pierre Karpman

`pierre.karpman@univ-grenoble-alpes.fr`
`https://membres-ljk.imag.fr/Pierre.Karpman/tea.html`

2021-09-29

Pointeurs : déclaration et manipulation

(Dés)Allocation

Pointeurs de fonction & compléments de type

Pointeurs en C : définition

Un pointeur `p` vers un type `type`, déclaré `type *p` est une variable contenant l'adresse d'une zone mémoire pouvant être peuplée d'un ou plusieurs éléments de type `type` :

```
uint8_t *p;
```

```
...
```

```
printf("%p\n", p);
```

```
...
```

donne par ex. :

```
0x7f8cf8c027c0
```

Le `*` peut être attaché au nom de la variable. On peut par ex. faire `int a, *b`; Mais il est attaché au type pour faire un cast :

```
a = (int *)b;
```

Exemples de pointeurs

Cas les plus courants en C :

Variables de type pointeur à la déclaration

```
uint8_t *p; // déclaration explicite
    float t[12]; // (pas tout à fait un pointeur)
    ↪ déclaration & allocation via la création
    ↪ d'un tableau
char *s = "HAI"; // (pas tout à fait un pointeur)
    ↪ chaîne de caractères (plus de détails dans
    ↪ quelques cours)
int **p2; // pointeur sur un pointeur sur un int
```

Exemples de pointeurs (2)

On peut aussi « créer » une expression de type pointeur en accédant à l'adresse d'une variable avec `&` :

```
uint8_t a;  
...  
printf("%p\n", &a);  
...
```

donne par ex. :

0x7ffee1d7a92b

→ Raison de la présence d'`&` dans les appels à `scanf`

Quel intérêt ?

Deux raisons fondamentales :

- ▶ Adresser de grandes zones mémoires (tout ne tient pas en registre...) à des emplacements déterminés dynamiquement
- ▶ Implémenter le passage d'arguments par référence

Mais la manipulation explicite de pointeurs est malheureusement la source de plein de bugs \rightsquigarrow mécanisme absent de nombreux langages de haut niveau

Adressage avec des pointeurs

L'accès à la zone mémoire pointée se fait via l'opérateur préfixe `*` :

```
a = *p; // copie la valeur stockée à l'adresse p dans a
*q = a; // écrit a à l'adresse q
```

On peut aussi accéder à une adresse relativement à un pointeur des façons suivantes :

```
a = *(p+7);
q[3] = a; // équivalent à *(q+3) = a;
b = *(p - 1); // à éviter
```

Mais il faut faire attention à ne pas faire d'accès hors-zone allouée ! (Comme pour les tableaux)

Type d'un pointeur & adressage relatif

Le type d'un pointeur détermine essentiellement le calcul des adresses relatives, l'unité du décalage étant la taille du type :

```
uint8_t *p;  
uint64_t *q;  
...  
printf("%p\t%p\n", p, p+1);  
printf("%p\t%p\n", q, q+1);  
...
```

donne par ex. :

```
0x7ff2824027c0  0x7ff2824027c1  
0x7ff282402bb0  0x7ff282402bb8
```

Il existe aussi un type générique `void *` (d'incrément 1, à éviter d'utiliser relativement)

Type d'un pointeur & adressage relatif (2)

Les conversions de type s'appliquent aux pointeurs, et ont l'effet attendu sur l'adressage :

```
uint8_t *p;  
uint64_t *q;  
...  
printf("%p\t%p\n", p, p+1);  
printf("%p\t%p\n", q, (uint8_t *)q+1);  
...
```

donne par ex. :

```
0x7fce024027c0  0x7fce024027c1  
0x7fce02402bb0  0x7fce02402bb1
```

Type d'un pointeur & adressage relatif (3)

Attention : la taille de la zone pointée par un pointeur ne fait pas partie de son type ! Ci-dessous, `b` et `d` ont le même type :

```
uint8_t a;  
uint8_t *b = &a;  
uint8_t c[29];  
uint8_t *d = c;
```

Il est syntaxiquement correct d'écrire

```
b[7] = 3;  
d[5] = 2;
```

Mais la première affectation donne lieu à un accès mémoire illégal

Type d'un pointeur & adressage relatif (4)

On peut aussi définir un pointeur vers une zone
« multidimensionnelle », par ex :

```
int (*t)[16]; // 16 est la seconde dimension, qui  
→ doit être connue à la compilation
```

...

```
t[i][j] = 3; // equiv. à *((int*)t+i*16+j) = 3;  
int (*s)[32][16]; // idem en 3D
```

...

```
s[i][j][k] = 4; // *((int*)s+i*32*16+j*16+k) = 4;
```

- ▶ La zone pointée est allouée en une fois, et donc contiguë en mémoire ; l'adressage multidimensionnel est juste de l'aide syntaxique
- ▶ On peut lire la valeur `t[i]` // = `t+i` (peu d'intérêt), mais une assignation `t[i] = malloc(...)` est illégale

Type d'un pointeur & adressage relatif (5)

On peut aussi définir un pointeur vers un pointeur (et itérer *ad lib.*) :

```
int **t; // pointeur vers un pointeur, ou  
↪ «tableau» de pointeur
```

...

```
int *tt = *t; // ou int *tt = t[0]
```

```
*tt = 3; // ou tt[0] = 3, ou t[0][0] = 3
```

...

- ▶ L'adressage multidimensionnel n'est plus juste de l'aide syntaxique, chaque [] correspond à un déréférencement
- ▶ On peut (et on doit) allouer chaque ligne, colonne etc. séparément

Type d'un pointeur & adressage relatif (6)

Allocation :

```
int **t = malloc(10*sizeof(int*)); // /\
↳ sizeof(int*); le tableau aura 10 lignes
t[0] = malloc(10*sizeof(int)); // la première ligne
↳ aura 10 colonnes
t[1] = malloc(5*sizeof(int)); // et la seconde 5...
...
int *s[10]; // variante avec première allocation sur
↳ la pile /\ différent de int (*s)[10] !
```

- ▶ Plus flexible, mais plus coûteux que l'utilisation d'un type multidimensionnel
- ▶ À utiliser surtout quand cette solution n'est pas possible

Valeur spéciale : `null`

- ▶ Un pointeur peut prendre une valeur « spéciale » `null` pour indiquer (de façon testable) qu'il ne pointe vers « rien »
- ▶ Il existe une macro `NULL` permettant de spécifier une telle valeur `null`, mais elle peut entraîner des problèmes subtils de portabilité en fonction du type qu'elle utilise
- ▶ Il est préférable d'utiliser `0` pour indiquer `null`

- ▶ Il existe un type `void*` pour pointer vers un type a priori inconnu
- ▶ On est obligé d'effectuer une conversion de type avant de pouvoir affecter dans la zone pointée (on ne peut rien affecter (ou même déclarer) de type `void`)

Exemple :

```
void *t;
```

```
...
```

```
*t = 3; // -> error: incomplete type 'void' is not  
↪ assignable
```

```
*(int*)t = 3; // okay
```

Généricité et `void*` (2)

Exemple d'utilisation : prototype de `memcpy`

```
void * memcpy(void *restrict dst, const void  
↪ *restrict src, size_t n);
```

- ▶ Un seul prototype quelque soient les types de `src` et `dst` (qui peuvent être différents) : `int *`, `double *`...
- ▶ (Pour la signification de `restrict`, cf. plus loin)

Passage par référence

Les pointeurs permettent aisément d'écrire des fonctions qui modifient un argument : ex. `scanf`

Une variable de type tableau (de taille fixe ou variable) n'est pas exactement un pointeur ; elle n'est notamment pas assignable

- ▶ `int t[1]; t = malloc(...);` ~>
error: array type 'int [1]' is not assignable
- ▶ `int t[a]; printf("%p\n", t++);` ~>
error: cannot increment value of type 'int [a]'

En pratique, hors allocation (cf. plus bas) pointeurs et tableaux se manipulent généralement de la même façon.

Pointeurs et tableaux (2)

Un tableau se *dégrade* aussi souvent naturellement en pointeur, par ex. dans un appel de fonction :

```
...
int t[10];
f(t);
...
void f(int t[10])
{
    printf("%p\n", t++); // okay
}
```

↪ Le nombre d'éléments d'un tableau passé en argument n'est pas connu au sein d'une fonction !

Déclaration d'arguments pointeurs/tableaux

Plusieurs options existent pour déclarer la signature d'une fonction prenant un tableau/pointeur en argument. Si celui-ci est de taille fixe (connue à la compilation) on peut par exemple faire :

```
void f(int *t);  
void f(int t[]); // équivalent au précédent  
void f(int t[10]); // équivalent au précédent,  
  ↪ «auto-documenté»  
void f(int t[static 10]); // t doit pointer vers  
  ↪ un tableau d'int d'au moins 10 éléments  
  ↪ (possible warning à la compilation si ce  
  ↪ n'est pas le cas)
```

Les deux dernières options documentent naturellement le code, et sont à préférer

Déclaration d'arguments pointeurs/tableaux (2)

Si un tableau en argument est de taille variable (inconnue à la compilation), celle-ci doit être passée en argument, par exemple comme :

```
void f(int *t, size_t nelem);  
void f(size_t nelem, int t[nelem]); // équivalent  
→ au précédent  
void f(int t[nelem], size_t nelem); // ne marche  
→ pas, nelem «inconnu»
```

Encore une fois, la seconde option documente naturellement le code et est à préférer

Déclaration d'arguments pointeurs

Il est bien sûr possible pour une fonction de prendre en argument un pointeur qui n'est pas « issu » d'un tableau, et qui par ex. pourrait être égal au pointeur nul `NULL`. Dans ce cas la syntaxe d'une déclaration est « évidente » :

```
void f(int *t);
```

Pointeurs : déclaration et manipulation

(Dés)Allocation

Pointeurs de fonction & compléments de type

Allocation mémoire

- ▶ Les pointeurs sont utiles pour accéder à de la mémoire pour lire ou stocker des données ; cette mémoire doit avoir été *allouée* par le système
- ▶ Une zone mémoire qui a été allouée doit être *libérée* une fois qu'elle n'est plus utilisée
- ▶ On distingue allocation *statique*, quand la quantité allouée est connue du compilateur, et allocation *dynamique* quand ce n'est pas le cas

La déclaration d'un tableau (de taille fixe) en C a un double effet :

- ▶ Déclaration de la variable du type tableau (\sim pointeur) approprié
- ▶ Allocation statique sur la *pile* (stack) de la mémoire demandée, et affectation de l'adresse de base dans la variable

Particularités de l'allocation sur la pile :

- ▶ La mémoire est libérée à la fin de l'exécution de la fonction
- ▶ La quantité allouable est relativement faible, inférieure (stricte) à la taille max. de la pile (sous UNIX, consultable pour une machine donnée via `ulimit -s`)
- ▶ Pas de mécanisme facile pour déterminer si une allocation a échoué

Allocation sur la pile (ex. 1)

```
int *bad_alloc()
{
    int t[2000];
    return t;
}
...
int *s = bad_alloc(); // l'espace alloué par
    ↪ bad_alloc() a immédiatement été désalloué
s[768] = 1; // accès illégal
...
```

Un compilateur moderne émettra un avertissement, par ex. :

```
badalloca.c:7:9: warning: address of stack memory
associated with local variable 't' returned
[-Wreturn-stack-address]
```

Allocation sur la pile (ex. 2)

...

```
int t[1000000000]; // le pointeur de pile sera trop  
↳ décrémente à l'appel de fonction et pointera  
↳ hors de la zone réservée à la pile  
// N'importe quelle instruction entraînant une  
↳ manipulation du pointeur de pile (par ex. un  
↳ appel de fonction) entraînera une erreur de  
↳ segmentation
```

...

- ▶ Il ne faut donc pas déclarer des tableaux de taille « déraisonnable »
- ▶ ... et faire particulièrement attention pour du code devant être portable/critique
- ▶ (des analyseurs de pile comme le “stackanalyzer” d'absint peuvent aider)

Allocation dynamique sur la pile

En C, il est aussi possible d'allouer dynamiquement sur la pile, via deux mécanismes :

- ▶ Utilisation d'`alloca()` (découragé car non standard et non portable)
- ▶ À partir de C99, via les tableaux de taille variable, par ex. :
`int t[dim];`

Intérêts et limites de l'allocation dynamique sur pile :

- ▶ Mêmes caractéristiques que l'allocation statique
- ▶ Potentiellement plus rapide que l'allocation sur le tas (cf. la suite) ; se fait par simple incrémentation/décrémentation du pointeur de pile
- ▶ Syntaxiquement simple dans le cas des tableaux à taille variable

En général, l'allocation dynamique de mémoire se fait sur le *tas* (heap)

- ▶ Celle-ci n'est pas libérée automatiquement (ce qui peut mener à des fuites mémoire)
- ▶ La quantité allouable est essentiellement limitée par la quantité physique disponible (RAM + swap)

Une fonction d'allocation couramment utilisée en C est `malloc()`, déclarée dans `stdlib.h`, de prototype :

```
void *malloc(size_t size);
```

Qui a pour effet :

- ▶ En cas de succès, d'allouer `size` octets de mémoire contiguë et de retourner un pointeur vers le « début » de la zone (ç-à-d que la zone allouée se trouve entre l'adresse de retour `ptr` et `(uint8_t*)ptr + (size - 1)`)
- ▶ En cas d'échec (par ex. car il n'y a pas suffisamment de mémoire disponible), de retourner `NULL`

Il existe aussi des variantes comme `calloc`, `realloc()`, `valloc()`...

En pratique :

- ▶ On souhaite allouer de la mémoire pour un type particulier
- ▶ La taille de ce type n'est pas forcément d'un octet

On procède donc typiquement ainsi :

```
uint64_t *p = (uint64_t  
↪ *)malloc(1000000*sizeof(uint64_t));
```

où `sizeof` est une construction du langage C qui renvoie la taille (en octets) du type passé en argument

- ▶ La mémoire allouée sur le tas par `malloc()` est persistante, et doit être explicitement libérée quand elle n'est plus utile
- ▶ Ceci se fait avec la fonction `free()` dont l'utilisation est simple :

```
free(p); // p doit être un pointeur vers une zone  
↳ préalablement allouée par malloc ou une fonction  
↳ similaire
```

Attention à :

- ▶ Ne jamais appeler `free()` sur de la mémoire allouée autrement (par ex. un tableau)
- ▶ Ne jamais appeler `free()` plusieurs fois pour la même zone

Aliasing

- ▶ Une déclaration ou utilisation de pointeur n'est pas forcément associée à une allocation (cf. `scanf`, `swap...`)
- ▶ L'*aliasing* entre des pointeurs consiste à en avoir plusieurs pointant vers la même zone mémoire
- ▶ On peut aliaser des pointeurs volontairement (par ex. lors d'un passage de tableau par référence), mais c'est aussi une source de bugs potentiels, par ex. :
 - ▶ désallocation multiple (double free)
 - ▶ accès à une zone qui a déjà été désallouée
- ▶ L'aliasing peut être dur à détecter ; il ne suffit pas de comparer la valeur des pointeurs (cf. par ex. `int t[12]`, `*s = t+3;`)

Exercices rapides

Que font ces instructions, et lesquelles peuvent poser problème ?

```
int t[256]; t+37 = 3;
```

```
int t[256]; *(&t[37]) = 3;
```

```
int *s; s[12] = 8;
```

```
int t[1024], *s = t+512; s[-511] = 2;
```

```
int t[1000000000000]; t[0] = 1;
```

```
int p = (int *)malloc(768*sizeof(int));
```

Écrivez une fonction `swap` qui permet d'échanger la valeur de deux variables de type quelconque, et donnez un exemple d'utilisation pour deux variables de type `int`

Troubleshooting

- ▶ Les accès mémoire bas-niveau en C sont la source de très nombreux bugs, souvent détectés via une «erreur de segmentation»
- ▶ Mais **Il n'y a pas équivalence entre accès mémoire illégaux et erreurs de segmentation**
- ▶ Un outil pratique pour détecter / résoudre des problèmes d'accès mémoire : valgrind

```
> valgrind ./a.out
...
==2564== Invalid write of size 4
==2564==    at 0x100000F2B: main (toto.c:8)
==2564==    Address 0x100dea800 is 0 bytes
↪   after a block of size 80 alloc'd
==2564==    at 0x1000AC086: malloc
==2564==    by 0x100000F0A: main (toto.c:5)
```

Pointeurs : déclaration et manipulation

(Dés)Allocation

Pointeurs de fonction & compléments de type

- ▶ En C, on peut faire référence à une fonction par son adresse, qui a un type pointeur

- ▶ L'adresse d'une fonction :

```
type_ret fun(type_a1 a1, type_a2 a2, ...)
```

peut être manipulée via un pointeur :

```
type_ret (*fun_ptr)(type_a1, type_a2, ...)
```

et est obtenue via `&fun` (ou simplement `fun`)

Exercice : QSORT(3)

Expliquez le prototype de la fonction `qsort` de la bibliothèque standard :

```
void qsort(void *base, size_t nel, size_t width, int  
↪ (*compar)(const void *, const void *));
```

Exercice : map

Écrivez une fonction `map` qui prend en entrée :

- ▶ Une liste d'`int` représentée par un tableau
- ▶ Une fonction de type `int -> int`

Et qui modifie la liste en appliquant la fonction sur chaque élément

Pointeurs const

- ▶ N'importe quel type pointeur peut être qualifié de `const`, par ex. pour donner `const int *p`
- ▶ Ceci interdit (à la compilation) la modification du contenu pointé par `p` :

```
void nocst(const int *p)
{
    p[0] = 1;

}
```

↪ error: read-only variable is not assignable

- ▶ Utile pour éviter les erreurs, pour la documentation, pour les pointeurs de fonction

- ▶ On peut tricher, au prix d'un warning :

```
void nocst(const int *p)
{
    int *q = p;
    q[0] = 1;

}
```

↪ warning: initializing 'int *' with an expression of type 'const int *' discards qualifiers

- ▶ On peut aussi utiliser `const` avec des types simples :
`const int a` \rightsquigarrow toute modification de `a` sera rejetée...
- ▶ Ou encore `int * const p` \rightsquigarrow on ne peut pas modifier le *pointeur* lui-même...
- ▶ Moins courant et moins utile, car sans effets de bords

Les variables locales `static`

- ▶ Une variable locale à une fonction peut être déclarée `static` afin d'être allouée une unique fois \rightsquigarrow « variable globale » visible uniquement dans la fonction
- ▶ Si la variable est initialisée à la déclaration (par ex. `static int a = 0`), cette initialisation n'est faite qu'une unique fois au début de l'exécution du programme
- ▶ Utile pour par ex. implémenter des compteurs de ressource

Exercice : écrire une fonction qui affiche le nombre de fois qu'elle a été appelée

Les variables globales `static`

- ▶ Une variable globale peut être déclarée `static` : ceci interdit son utilisation depuis un autre fichier `.c/.o`

```
// a.c
```

```
static int toto = 1;
```

```
// b.c
```

```
extern int toto; // cherche à référencer une variable
```

```
↳ déclarée dans un autre fichier -> celle-ci n'est
```

```
↳ pas trouvée ici car elle a été déclarée 'static'
```

```
...
```

```
printf("%d\n", toto);
```

```
Undefined symbols for architecture x86_64:
```

```
  "_toto", referenced from:
```

```
    _main in tm.o
```

```
ld: symbol(s) not found for architecture x86_64
```

Un autre sens pour `static`

- ▶ Un tableau/pointeur argument de fonction peut être déclaré comme (par ex.) :

```
void f(int t[static 5]);
```

- ▶ Indique que `t` doit contenir *au moins* 5 éléments
- ▶ Utile à des fins de documentation
- ▶ Et permet au compilateur de détecter certains mauvais appels à la fonction

Quelques autres qualificatifs

- ▶ `volatile` : une variable `volatile` peut être modifiée par un phénomène extérieur
- ▶ `register` : une variable `register` ne peut pas être déréférencée (possiblement utile pour détecter des prises d'adresses inadvertantes)
- ▶ `restrict` : déclare l'intention de ne pas effectuer d'aliasing (utile à des fins de documentation, par ex. `memcpy` plus haut)

Types spéciaux (1) : struct

- ▶ Il est possible en C d'agglomérer plusieurs types en un seul grâce à `struct`, par ex :

```
struct misc
{
    int a;
    double b;
    unsigned t[160];
    char *s;
    int (*f) (int, int);
};
```

- ▶ Utilisation : définition de la structure : comme ci-dessus ;
déclaration de variables : `struct misc a;` ; accès aux
composants : `a.a`, `a.s[7]`...

Types spéciaux (1) : `struct` (2)

Quelques règles sur les structures :

- ▶ Un champ d'une structure peut être une autre structure
- ▶ Un champ d'une structure peut être un *pointeur* vers la structure étant définie
- ▶ La taille en mémoire d'une structure peut s'obtenir avec `sizeof`
- ▶ Le passage de structures en argument se fait par valeur

Il existe aussi une notation concise pour les pointeurs vers des `struct` : `(*s).a` peut se remplacer par `s->a`

Types spéciaux (1) : struct (3)

Quelques intérêts des `structs` :

- ▶ Utile pour la définition de structures de données récursives, par ex :

```
struct bt
{
    struct bt *left;
    struct bt *right;
    size_t size;
    void *data;
};
```

- ▶ Utile pour regrouper des données logiquement liées entre elles
↪ on peut « facilement » faire de la P.O. en C avec des `structs` et des pointeurs de fonction

Types spéciaux (1) : struct (4)

On peut initialiser une structure à la déclaration, « comme » un type non composé :

```
struct s
{
    int a;
    int b;
};
...
struct s X = {.a = 1, .b = 1};
...
```

Types spéciaux (2) : enum

On peut utiliser un type `enum` pour associer un nom à une valeur entière, par ex. :

```
enum status
{
    on,
    off,
};
enum status a = on;
if (a == 0) { a += off; }
```

↪ Aucune contrainte de typage, purement « visuel »

Types spéciaux (3) : union (1)

Un type `union` sert à référencer un emplacement mémoire avec différents types \rightsquigarrow comme une `struct`, mais les emplacements mémoire se chevauchent. Ex. :

```
union uint64d
{
    double d;
    uint64_t i;
};
union uint64d x;
x.d = M_SQRT2;
```

permet d'accéder à la représentation binaire (mantisse et exposant) du `double` `M_SQRT2` via `x.i`

Types spéciaux (3) : union (2)

Autre exemple :

```
union uint64st
{
    uint64_t i;
    uint8_t t[8];
};
union uint64st x;
x.i = 0x0123456789ABCDEF;
```

permet d'accéder aux octets de `x.i` via `x.t[]` (dans ce cas, un résultat similaire pourrait être obtenu par cast de pointeurs)

Exercice de taille

Soit les déclarations suivantes :

```
union u1
{
    uint64_t a;
    uint64_t b;
    uint64_t c;
};
struct s1
{
    uint64_t a;
    uint64_t b;
    uint64_t c;
};
```

Combien valent `sizeof(union u1)` et `sizeof(struct s1)` au minimum ?

Le renommage typedef

- ▶ Pour plus de confort, on peut donner un *alias* à un type *existant* avec une déclaration `typedef type alias`
- ▶ Souvent utilisé pour raccourcir les déclarations de types construits, par ex.

```
typedef union uint64d u64d;  
u64d x; // au lieu de union uint64d x  
...
```

- ▶ Il vaut mieux éviter de renommer les types standards comme

```
typedef uint8_t petit_poney;  
typedef uint16_t moyen_poney;  
typedef uint8_t *pointeur_petit_poney; // horrible  
...
```

- ▶ À utiliser avec une certaine parcimonie...