C 8 : pointeurs et allocation dynamique

Juliusz Chroboczek

22 novembre 2023

1 L-valeurs

On appelle *l-valeur (l-value, left-value)* une construction qui peut apparaître à la gauche d'un symbole d'affectation « = ». Ce sont notamment :

- les noms de variable ;
- les paramètres formels de fonctions ;
- les cases de tableau ;
- les champs de structure.

En cours, j'ai dessiné les l-valeurs comme des rectangles.

Les valeurs ordinaires obéissent au principe de substitution : si $x = y$, alors on peut substituer y à x. Par exemple, comme $3 + 2 = 5$, on peut remplacer $3 + 2$ par 5 partout où il apparaît. Ce n'est pas le cas des l-valeurs : même si `x == y`, il n'est en général pas correct de remplacer l'instruction `x = 1` par `y = 1`.

2 Pointeurs

Une *référence* est une valeur ordinaire qui identifie une l-valeur. Si les l-valeurs sont des rectangles, les références sont des flèches qui pointent vers les rectangles. En C, les références sont implémentées par des *pointeurs*. Un pointeur est tout simplement une adresse de mémoire.

Si T est un type, les pointeurs sur les l-valeurs dans T ont le type T`*`. Par exemple, « `int *` » est le type des pointeurs sur les entiers.

Chaque type pointeur contient la valeur `NULL`, une valeur distinguée qui ne pointe sur aucune l-valeur. La seule chose qu'on peut faire avec un pointeur valant `NULL`, c'est le comparer à un autre pointeur.

Il existe deux opérations principales sur les pointeurs. L'opérateur « `&` » crée une référence à une l-valeur : si x est une l-valeur, alors `&`x est un pointeur qui réfère à x. Son inverse est l'opérateur « `*` » qui *déréférence* un pointeur : si p réfère à x, alors `*`p est équivalent à x. Dans ce cours, nous éviterons d'utiliser l'opérateur « `&` » sauf lors des appels aux fonctions (voir ci-dessous).

2.1 Pointeurs et structures

Les pointeurs sur les structures se comportent exactement comme les pointeurs sur les valeurs de base.

```
struct valeur {
    int v;
};
...
struct valeur s;
struct valeur *p;
p = &s;
(*p).v = 42;
printf("%d\n", (*p).v);
```

Si *p* est un pointeur sur une structure, alors *p*`->v` est une abbréviation pour `(*`*p*`).v` :

```
printf("%d\n", p->v);
```

2.2 Pointeurs et tableaux

S'il est possible de créer un pointeur sur un tableau, nous ne le ferons pas dans ce cours. À la place, nous manipulerons les pointeurs sur le premier élément d'un tableau :

```
int a[100];
int *p;
p = &(a[0]);
```

Ceci est rendu plus facile par deux particularités du C :

- si *a* est un tableau, alors *a* peut être *coercé* en un pointeur sur son premier élément ;
- si *p* est un pointeur sur un élément d'un tableau, alors il peut être indexé : si *p* = `&(a[0])`, alors *p*`[n]` est équivalent à `a[n]`.

Le code ci-dessus est équivalent à

```
int a[100];
int *p;
p = a;
```

et on peut accéder à un élément arbitraire de `a` à travers `p` :

```
p[42] = 15;
printf("%d\n", a[42]);      /* affiche 15 */
```

2.3 Pointeurs et fonctions

Un pointeur peut être passé par valeur à une fonction. La fonction peut alors modifier le contenu de la l-valeur à laquelle réfère le pointeur à travers ce dernier.

```
void f(int x) { x = 42; }
void g(int *x) { *x = 42; }

int v;
v = 12;
f(v);
printf("%d\n", v);       /* affiche 12 */
g(&v);
printf("%d\n", v);       /* affiche 42 */
```

Nous avons déjà utilisé cette technique lorsque nous avons appelé la fonction `scanf`, et aussi lorsque nous avons passé un (pointeur sur le premier élément d'un) tableau à une fonction.

2.4 Pointeurs non-typés

Le type `void *` représente les pointeurs non-typés. Un pointeur non-typé doit être converti en un pointeur typé avant utilisation, et c'est la responsabilité du programmeur que de choisir le bon type de destination.

3 Allocation dynamique

Nous avons déjà vu deux types d'allocation de mémoire :

- l'allocation *statique*, qui s'applique aux variables définies en dehors de toute fonction ;
- l'allocation sur la pile, ou allocation *automatique*, qui s'applique aux variables définies dans une fonction.

L'allocation statique se fait lors de la compilation, et alloue des données dans les sections *data* et *bss* du processus ; leur taille peut être déterminée statiquement (lors de la compilation). A contrario, l'allocation automatique se fait lors d'un appel à une fonction, et les données sont stockées sur la pile *(stack)*. Les allocations automatiques obéissent à la discipline de pile : si a est alloué avant b, alors b sera forcément libéré avant a.

Il existe un troisième type d'allocation : l'allocation dynamique, qui se fait dans une zone de mémoire appelée *tas (heap)*. Comme une allocation automatique, une allocation dynamique se fait à l'exécution. À la différence de l'allocation automatique, elle n'obéit pas à la discipline de pile — même si a est alloué avant b, il est possible de libérer a avant b.

Une allocation dynamique se fait à l'aide des fonction `malloc` et `free` :

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

Le type `size_t` est un synonyme pour un type entier non-signé, typiquement `unsigned int` ou `unsigned long int`.

La fonction `malloc` alloue un bloc de mémoire d'une taille égale à `size`; ce paramètre est normalement calculé à l'aide de l'opérateur `sizeof`, qui retourne la taille en octets du type qui lui est passé en paramètre. Par exemple, `sizeof(int)` est la taille d'un entier (typiquement 4) :

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

alloue 4 octets de mémoire dynamique et y fait référer le pointeur `p`. La fonction `malloc` peut échouer s'il n'y a pas assez de mémoire; elle retourne alors `NULL`.

La fonction `free` libère un bloc de mémoire alloué à l'aide de `malloc` : le pointeur passé à `free` devient invalide, et c'est la responsabilité du programmeur que de s'assurer qu'il n'est plus utilisé. Par exemple, l'allocation ci-dessus peut être libérée à l'aide de l'appel suivant :

```
free(p);
```

L'allocation dynamique en C laisse beaucoup de responsabilités au programmeur [1]. Le programme `valgrind` est capable d'identifier la plupart des violations de cette règle. Il est *obligatoire* de tester tous vos programmes qui font des allocations dynamiques à l'aide de `valgrind`.

3.1 Allocation dynamique des tableaux

Un tableau de taille n occuppe n fois plus de mémoire qu'une valeur de son type de base; on calcule donc son occuppation de mémoire à l'aide d'une multiplication :

```
int *p;
p = malloc(42 * sizeof(int));
/* p[0] à p[41] sont maintenant valides */
free(p);
/* p est maintenant empoisonné */
```

3.2 Fonctions utilitaires

La fonction `memset` permet d'initialiser à 0 une zone de mémoire. Elle consiste d'une boucle qui parcourt une zone de mémoire et affecte 0 à tous les octets qui la constituent.

```
int *p;
p = malloc(42 * sizeof(int));
memset(p, 0, 42 * sizeof(int));
```

La fonction `calloc` combine `malloc` avec `memset` pour allouer une zone de mémoire initialisée à 0; pour des raisons historiques, elle prend deux paramètres, le nombre d'éléments à allouer et la taille d'un élément. Elle pourrait être écrite comme suit :

1. Les langages modernes, comme Java ou Go, utilisent un *grabage collector* qui évite au programmeur de devoir libérer la mémoire à l'aide de `free`.

```
void *calloc(size_t nmemb, size_t size) {
    void *p;
    p = malloc(nmemb * size);
    if(p == NULL)
        return NULL;
    memset(p, 0, nmemb * size);
    return p;
}
```

La fonction `memmove` permet de recopier une zone de mémoire.

```
int a[42];
int *p;
for(i = 0; i < 42; i++)
    a[i] = 57 * i;
p = malloc(42 * sizeof(int));
memmove(p, a, 42 * sizeof(int));
/* p contient maintenant une copie des données de a */
```

La fonction `memcpy` est une variante optimisée de `memmove` qui ne fonctionne correctement que lorsque les zones de mémoire source et destination ne se recouvrent pas. Il n'est pas nécessaire de s'en servir, elle n'est que marginalement plus rapide que `memmove`.