C 10 : tableaux à deux dimensions

Juliusz Chroboczek

29 novembre 2023

Les tableaux que nous avons vus précédemment sont à *une dimension* : ils sont indexés par des entiers, et sont isomorphes aux suites finies ou aux vecteurs en mathématiques. Les tableaux à *deux dimensions* sont indexés par deux entiers, et correspondent donc aux *matrices* en mathématiques.

(a) (b)

FIGURE 1 — Tableaux à une (a) et deux (b) dimensions

Il existe deux façons d'implémenter les tableaux à deux dimensions en C : de façon contiguë et comme des tableaux de tableaux.

1 Allocation contiguë

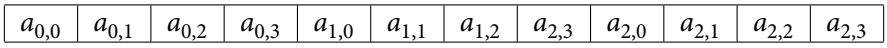

FIGURE 2 — Un tableau à deux dimensions alloué de façon contiguë

On peut représenter un tableau à deux dimensions par un tableau à une dimension (figure 2). Un tableau A de m lignes et n colonnes est représenté par un tableau a de $m \times n$ cases, et l'élément $A_{i,j}$ est stocké dans a_{ni+j}.

Cette approche est facile à implémenter en C ; par exemple, la fonction suivante crée une table de multiplication :

```
int *table() {
    int *a = malloc(10 * 10 * sizeof(int));
    if(a == NULL)
```

```
        return NULL;

    for(int i = 0; i < 10; i++) {
        for(int j = 0; j < 10; j++) {
            a[10 * i + j] = (i + 1) * (j + 1);
        }
    }
    return a;
}
```

Cette approche est très efficace du moment que l'on prend soin de parcourir les données dans l'ordre dans lequel elles stockées en mémoire. Dans le fragment de code ci-dessous, j'ai parcouru la table de multiplication ligne par ligne justement dans le but de causer des accès séquentiels à la mémoire.

Digression : tableaux multidimensionnels natifs Le C sait gérer nativement les tableaux à deux dimensions. Un tableau à deux dimensions natif est déclaré avec la syntaxe

```
int a[lignes][colonnes];
```

et on accède à ses éléments à l'aide de `a[`*i*`][`*j*`]`. Cependant, ces tableaux à deux dimensions ne se combinent pas facilement avec les techniques de type *tableaux Pascal* vues en TP, je ne m'en sers donc pratiquement jamais.

2 Tableaux de tableaux

Un tableau à deux dimensions peut aussi être représenté par un tableau de pointeurs sur des tableaux, donc en stockant les lignes de façon discontiguë en mémoire. Une telle représentation est un peu moins efficace, car les données ne sont pas contiguës et il faut y accéder à travers des pointeurs. Par contre, elle a l'avantage d'être plus pratique à manipuler, et de se combiner facilement avec les techniques de type *tableaux Pascal* et *slices*.

Dans cette représentation, on manipule des pointeurs sur des pointeurs ; par exemple, si les éléments sont de type `int`, le tableau sera représenté par une valeur de type « `int **` ». L'exemple suivant alloue une table de multiplication représentée de façon discontiguë :

```
int **table2() {
    int **a = malloc(10 * sizeof(int*));
    if(a == NULL)
        return NULL;

    for(int i = 0; i < 10; i++) {
        a[i] = malloc(10 * sizeof(int));
        if(a[i] == NULL) {
            for(int k = 0; k < i; k++)
                free(a[k]);
```

```
            free(a);
            return NULL;
        }
    }

    for(int i = 0; i < 10; i++) {
        for(int j = 0; j < 10; j++) {
            a[i][j] = (i + 1) * (j + 1);
        }
    }
    return a;
}
```

Vous remarquerez que l'allocation discontiguë complique la gestion d'erreurs — au moment où un `malloc` échoue, il faut défaire les allocations précédentes. En revanche, elle simplifie l'accès aux éléments du tableau.

En général, il vaut mieux utiliser une approche de type *tableau Pascal* et encapsuler le tableau et ses bornes à l'intérieur d'une seule structure :

```
struct array {
    int **a;
    int lines, cols;
}
```

Le code suivant alloue une table de multiplication dans une telle structure :

```
struct array *table2() {
    struct array *a = malloc(sizeof(struct array));
    if(a == NULL)
        return NULL;

    a->lines = 10;
    a->cols = 10;

    a->a = malloc(a->lines * sizeof(int*));
    if(a->a == NULL) {
        free(a);
        return NULL;
    }

    for(int i = 0; i < a->lines; i++) {
        a->a[i] = malloc(10 * sizeof(int));
        if(a->a[i] == NULL) {
            for(int k = 0; k < i; k++)
                free(a->a[k]);
```

```
            free(a->a);
            free(a);
            return NULL;
        }
    }

    for(int i = 0; i < a->lines; i++) {
        for(int j = 0; j < a->cols; j++) {
            a->a[i][j] = (i + 1) * (j + 1);
        }
    }
    return a;
}
```