# Permutation qui ordonne en tableau, en C

# Michel Billaud (michel.billaud@laposte.net)

### 2022-03-01

# Table des matières

1	Uti	lisation de qsort_r
2	Uti	lisation de qsort
3		nment écrire un tri réentrant
	3.1	Tri d'un tableau d'entiers
	3.2	Tri paramétré d'un tableau d'entiers
	3.3	Version "réentrante"
	3.4	Et ensuite?

Ce texte fait partie d'une petite collection de notes mise à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 2.0 France.

- Les notes sont publiées dans https://www.mbillaud.fr/notes/
- Sources dans https://github.com/MichelBillaud/notes-diverses

Dernières corrections : 1er mars 2022

#### Résumé

On veut trouver la permutation qui ordonne un tableau.

Exemple: pour le tableau

```
int array[9] = { 66, 11, 44, 22, 88, 55, 77, 99, 33};
la suite d'indices {1, 3, 8, 2, 5, 0, 6, 4, 7}
parce que array[1]=11, array[3]=22, array[8]=33, etc.
```

- 1. C'est facile à faire avec qsort\_r qui est une extension GNU.
- Un bricolage permet de le faire avec qsort, mais ça donne du code non réentrant.
- 3. On montre comment adapter un algo de tri en fonction de tri "réentrante" paramétrée par une fonction de comparaison.

# 1 Utilisation de qsort\_r

Trouver la (enfin, une) permutation qui ordonne un tableau, on peut le faire facilement avec le fonction  $qsort_r$ :

```
#define _GNU_SOURCE 1
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
int comparator_r(const void *v1, const void *v2, void *context) {
    const int *p1 = v1, *p2 = v2;
    int *array = context;
    return array[*p1] - array[*p2];
void test_qsort_r()
    int array[9] = { 66, 11, 44, 22, 88, 55, 77, 99, 33};
    int perm[9] = { 0, 1, 2, 3, 4, 5, 6, 7, 8};
    qsort_r(perm, 9, sizeof(array[0]), comparator_r, array);
    for (int i = 0; i < 9; i++) {
       printf("[%d]=%d ", perm[i], array[perm[i]]);
    printf("\n");
}
int main()
{
   test_qsort();
    return 0;
```

parce qu'il faut à la fois passer le tableau à trier (la permutation), et le tableau de valeurs.

#### Exécution

```
[1]=11 [3]=22 [8]=33 [2]=44 [5]=55 [0]=66 [6]=77 [4]=88 [7]=99
```

Mais bon, c'est une extension GNU, donc pas conforme aux standards de la bibliothèque C, ni POSIX.

# 2 Utilisation de qsort

On peut le faire avec **qsort** : au lieu de faire du "contexte" un paramètre du comparateur, on utilise dans le comparateur une variable globale :

```
void * comparator_context;
```

```
int comparator(const void *v1, const void *v2) {
    const int *p1 = v1, *p2 = v2;
    int *array = comparator_context;
    return array[*p1] - array[*p2];
}

void test_qsort()
{
    int array[9] = { 66, 11, 44, 22, 88, 55, 77, 99, 33};
    int perm[9] = { 0, 1, 2, 3, 4, 5, 6, 7, 8};
    comparator_context = array;
    qsort(perm, 9, sizeof(array[0]), comparator);

for (int i = 0; i < 9; i++) {
        printf("[%d]=%d ", perm[i], array[perm[i]]);
    }
    printf("\n");
}</pre>
```

ce qui est typiquement le genre de choses malpropre à éviter parce qu'elles conduisent à du code non réentrant (si plusieurs threads font des tris en même temps, ça va pas le faire).

C'est sans doute pour ça qu'ils ont appelé qsort\_r, avec le suffixe habituel des fonctions réentrantes qui ont été ajoutées (exemple strtok\_r).

Par elle-même, la fonction standard qsort est réentrante, mais si on veut l'utiliser on est souvent obligé de faire un hack qui ne l'est pas. La fonction qsort\_r évite ça.

## 3 Comment écrire un tri réentrant

Dans cette partie, on regarde les étapes qui mènent d'un algo de tri à une fonction qui fait un tri réentrant, à la manière de qsort\_r.

Pour faire simple, on se base sur le tri par sélection :

- on parcourt le tableau pour trouver le plus petit élément, que l'on place dans la première case,
- on parcourt le tableau à partir de la 2ieme case, on amène le minimum en 2ieme position
- etc

Il est bien connu que ce n'est pas un tri performant : son temps d'exécution est quadratique (proportionnel au carré du nombre de valeurs à trier). C'est juste un exemple.

### 3.1 Tri d'un tableau d'entiers

Commençons tout d'abord par le tri croissant d'un tableau d'entiers :

```
void selint_sort(int array[], size_t size)
{
```

```
for (size_t i = 0; i < size; i++) {</pre>
        size_t m = i;
        for (size_t j = i + 1; j < size; j++) {</pre>
             if (array[j] < array[m] ) {</pre>
                 m = j;
        }
        if (m != i) {
             int tmp = array[i];
             array[i] = array[m];
             array[m] = tmp;
    }
}
La fonction de test montre comment l'utiliser
void test_selint()
    printf("# test selint\n");
    int array[9] = { 66, 11, 44, 22, 88, 55, 77, 99, 33};
    comparator_context = array;
    selint_sort(array, 9);
    for (int i = 0; i < 9; i++) {
        printf("%d ", array[i]);
    printf("\n");
et l'exécution montre sans surprise qu'on obtient le résultat attendu :
# test selint
11 22 33 44 55 66 77 88 99
```

## 3.2 Tri paramétré d'un tableau d'entiers

Un paramètre supplémentaire indique la fonction de comparaison entre entiers

```
}
}
}
```

Ici on se limite a un tri d'entiers (int) : on se contente donc de passer des valeurs entières à la fonction de comparaison, là où qsort passer des adresses pour assurer la généricité.

Exemple d'utilisation : tri d'un tableau dans l'ordre décroissant

## 3.3 Version "réentrante"

La version réentrante prend en paramètre supplémentaire un "contexte" qu'elle transmet au comparateur

```
}
}
Utilisation pour trouver la permutation qui ordonne un tableau d'entiers dans
l'ordre croissant :
int comp_r(const int i1, const int i2, void *context) {
    int *array = context;
    return array[i1] - array[i2];
}
void test_selint_sort_r()
    printf("# test_selint_sort_r\n");
    int array[9] = { 66, 11, 44, 22, 88, 55, 77, 99, 33};
    int perm[9] = { 0, 1, 2, 3, 4, 5, 6, 7, 8};
    selint_sort_r(perm, 9, comp_r, array);
    for (int i = 0; i < 9; i++) {
        printf("[%d]=%d ", perm[i], array[perm[i]]);
    printf("\n");
}
Résultat
# test_selint_sort_r
[1]=11 [3]=22 [8]=33 [2]=44 [5]=55 [0]=66 [6]=77 [4]=88 [7]=99
```

### 3.4 Et ensuite?

Ensuite, on pourrait rendre ce tri générique, avec un prototype semblable à celui de  ${\tt qsort\_r}$