C - Passage de paramètres par valeur et par référence

Quand on appelle une fonction en C, on peut lui **transmettre des valeurs** à traiter. Mais il y a uen différence importante à prendre en compte selon que tu veuilles modifier ces valeurs ou pas.

5TTR 6TTR Découverte

Introduction

Selon la manière dont on transmet les valeur à une fonction, celle-ci peut :

- soit **travailler sur une copie** de la variable (sans modifier l'originale),
- soit **modifier directement la variable d'origine**.

C'est la différence entre le **passage par valeur** et le **passage par référence**.

Objectifs

À la fin de ce chapitre, tu sauras :

- Faire la différence entre **passage par valeur** et **passage par référence**.
- Comprendre pourquoi on utilise `&` pour récupérer l'adresse d'une variable.
- Identifier les **types automatiquement passés par référence**.
- Savoir quand utiliser l'une ou l'autre méthode.

Passage par valeur

Principe

Lorsqu'une variable est passée **par valeur**, la fonction reçoit **une copie** de cette variable. Les modifications faites dans la fonction **ne changent pas** la variable originale.

Exemple

```c
#include <stdio.h>

void incrementer(int x) {
    x = x + 1;  // modifie seulement la copie
}

int main() {
    int nombre = 10;
    incrementer(nombre);
    printf("Valeur après appel : %d\n", nombre);
    return 0;
}
```

Résultat

```
Valeur après appel : 10
```

La fonction a modifié **la copie**, jamais la variable d'origine.

Passage par référence

Principe

Dans ce cas, la fonction reçoit **l'adresse mémoire** de la variable, grâce au symbole `&`. Elle peut donc **modifier directement la valeur originale**.

Exemple

```c
#include <stdio.h>

void incrementer(int *x) {
    *x = *x + 1;  // modifie la véritable variable
}

int main() {
    int nombre = 10;
    incrementer(&nombre);
    printf("Valeur après appel : %d\n", nombre);
    return 0;
}
```

Résultat

```
Valeur après appel : 11
```

Types passés automatiquement par référence

En C, certains types sont **toujours passés par référence**, même si tu ne mets pas explicitement de pointeur. Ce n'est pas une règle magique du langage, mais une conséquence directe de la façon dont ces types sont **stockés en mémoire**.

Les tableaux (`int tab[]`, `char nom[]`, etc.)

Quand tu passes un tableau, la fonction reçoit **l'adresse du premier élément**. Le tableau **n'est jamais copié**.

```c
void afficherPremier(int t[]) {
    printf("%d\n", t[0]);
}

int main() {
    int valeurs[3] = {4, 7, 2};
    afficherPremier(valeurs); // pas besoin de &
    return 0;
}
```

➡️ Toute modification dans la fonction modifie le tableau d'origine.

Les chaînes de caractères (`char nom[]`)

Une chaîne est un **tableau de char**, donc même logique :

```c
void mettreEnMaj(char s[]) {
    s[0] = 'A';   // modifie l'original
}
```

➡️ Aucun `&` nécessaire, la chaîne est déjà une adresse.

Les pointeurs

Si tu passes un pointeur, tu passes déjà une **adresse mémoire**.

```c
void changer(int *p) {
    *p = 99;
```

```
3 | }
```

➡️ Directement par référence.

Comparaison des deux approches

Mode de passage	Exemple d'appel	Modifie la variable originale ?
Par valeur	`incrementer(x)`	❌ non
Par référence (via pointeur)	`incrementer(&x)`	✔ oui
Tableau	`fonction(tab)`	✔ oui (toujours)
Chaîne de caractères	`fonction(nom)`	✔ oui (toujours)

À retenir

- **Par valeur** → la fonction reçoit une **copie** → la variable originale ne change pas.
- **Par référence** → la fonction reçoit **une adresse** → la variable originale peut changer.
- Les **tableaux** et **chaînes de caractères** sont **toujours** transmis par référence.
- On utilise `*` et `&` pour manipuler les adresses et les valeurs pointées.