ChatGPT

Pointeurs et constance en C

Un pointeur en C est une variable qui contient l'adresse d'une autre variable. La **constance** permet de déclarer que certaines données ne peuvent pas être modifiées. Par exemple :
- `const int *p;` : `p` est un pointeur vers un **int constant**. On peut modifier `p` (pointer vers une autre adresse), mais on **ne peut pas** modifier `*p`.
- `int *const p;` : `p` est un **pointeur constant vers un int**. On ne peut pas modifier `p` lui-même (il est fixé à une adresse), mais on peut modifier la valeur pointée `*p`.
- `const int *const p;` : `p` est un **pointeur constant vers un int constant**. Ni le pointeur `p`, ni la valeur pointée `*p` ne peuvent être modifiés.

Pour **lire une déclaration** de pointeur, on commence au nom du pointeur et on lit d'abord à droite si possible, sinon à gauche. Par exemple `int *const p` se lit « `p` est un pointeur constant vers un entier ».

On peut utiliser `typedef` pour simplifier les types. Par exemple :

```
typedef int MyInt;
typedef MyInt *IntPtr;
const IntPtr p; // équivaut à 'int *const p' : pointeur constant vers int
```

Bonnes pratiques :
- Préférez écrire `const int *p` plutôt que `int const *p` (évitez le style dit *"east const"*). Cela rend la lecture plus claire (on déclare d'abord `const`).
- Si vous allouez de la mémoire dynamique (avec `malloc`), n'oubliez pas de la libérer (`free`) pour éviter les fuites de mémoire.
- **Ne pas utiliser** `assert` pour valider les entrées utilisateur : `assert` sert au débogage interne, pas à la gestion d'erreur utilisateur.

Recherche linéaire dans un tableau

La **recherche linéaire** consiste à parcourir un tableau élément par élément pour trouver une valeur donnée, sans modifier le tableau original. On utilise un pointeur `const` pour garantir que l'on ne change pas les données.

Pseudocode de la recherche linéaire :
- Pour `i` de 0 à `length-1` :
- Si `array[i] == target`, retourner `i`.
- Si l'on n'a rien trouvé, retourner `length` (indice hors du tableau) pour indiquer « pas trouvé ».

Implémentation en C :

```c
int recherche_lineaire(const int *array, int length, int target) {
    for (int i = 0; i < length; i++) {
        if (array[i] == target) {
            return i;   // élément trouvé à l'indice i
        }
    }
    return length;  // élément non trouvé (retourne un indice hors tableau)
}
```

Cette fonction parcourt le tableau de gauche à droite. Comme `array` est un pointeur `const`, le tableau d'origine n'est pas modifié. On renvoie `length` pour signaler l'absence de `target`.

Recherche du minimum et du maximum en un seul parcours

On peut trouver simultanément le **minimum** et le **maximum** d'un tableau en une seule passe, plutôt que de faire deux parcours séparés.

Pseudocode :
- Initialiser `min = array[0]` et `max = array[0]`.
- Pour `i` allant de 1 à `length-1` :
- Si `array[i] < min`, alors `min = array[i]`.
- Si `array[i] > max`, alors `max = array[i]`.
- Retourner les deux valeurs `min` et `max`.

Implémentation en C :

```c
#include <limits.h>  // pour INT_MIN et INT_MAX

void min_max(const int *array, int length, int *min, int *max) {
    if (length <= 0) return;
    *min = *max = array[0];
    for (int i = 1; i < length; i++) {
        if (array[i] < *min) {
            *min = array[i];
        }
        if (array[i] > *max) {
            *max = array[i];
        }
    }
}
```

Cette fonction lit le tableau `array` (const) une seule fois. Les valeurs minimales et maximales sont renvoyées par pointeurs `min` et `max`.

Mélange de Fisher-Yates (Shuffle aléatoire)

L'**algorithme de Fisher-Yates** réalise un mélange aléatoire uniforme d'un tableau. Il parcourt le tableau de la fin vers le début, échangeant chaque élément avec un autre aléatoire parmi ceux non encore traités.

Pseudocode de Fisher-Yates :
- Appeler `srand(time(NULL))` une seule fois au début pour initialiser le générateur de nombres aléatoires.
- Pour `i` allant de `n-1` jusqu'à `1` :
- Tirer un indice `j` aléatoire dans l'intervalle `[0, i]` (`j = rand() % (i+1)`).
- Échanger `array[i]` et `array[j]`.

Implémentation en C :

```c
#include <stdlib.h>
#include <time.h>

void shuffle(int *array, int n) {
    srand((unsigned) time(NULL)); // initialisation du hasard (une seule
fois)
    for (int i = n - 1; i > 0; i--) {
        int j = rand() % (i + 1);
        // échange des éléments array[i] et array[j]
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

Chaque élément a alors une chance uniforme d'atterrir à n'importe quelle position. L'algorithme garantit un mélange bien réparti (chaque permutation est équiprobable).

Tri par sélection (Selection sort)

Le **tri par sélection** construit le tableau trié en échangeant à chaque itération l'élément le plus petit de la partie non triée vers la position courante.

Pseudocode :
- Pour `i` de 0 à `n-2` :
- `min_idx = i`
- Pour `j` de `i+1` à `n-1` :
- Si `array[j] < array[min_idx]`, alors `min_idx = j`.
- Échanger `array[i]` et `array[min_idx]`.

Implémentation en C :

```c
void tri_selection(int *array, int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < n; j++) {
            if (array[j] < array[min_idx]) {
                min_idx = j;
            }
        }
        if (min_idx != i) {
            int temp = array[i];
            array[i] = array[min_idx];
            array[min_idx] = temp;
        }
    }
}
```

- **Complexité** : en moyenne et au pire, environ `n*(n-1)/2` comparaisons et jusqu'à `n-1` échanges.
- **Performance réelle** : peu de swaps (au plus n-1), mais toujours `O(n^2)` comparaisons.

Tri par insertion (Insertion sort)

Le **tri par insertion** construit progressivement un tableau trié sur la gauche. À chaque étape, on insère le nouvel élément dans la bonne position parmi les éléments déjà triés.

Pseudocode :
- Pour `i` de 1 à `n-1` :
- `key = array[i]` ; `j = i - 1`.
- Tant que `j >= 0` et `array[j] > key` :
- Déplacer `array[j]` vers `array[j+1]`.
- `j--`.
- Placer `key` à `array[j+1]`.

Implémentation en C :

```c
void tri_insertion(int *array, int n) {
    for (int i = 1; i < n; i++) {
        int key = array[i];
        int j = i - 1;
        while (j >= 0 && array[j] > key) {
            array[j + 1] = array[j];
            j--;
        }
        array[j + 1] = key;
    }
}
```

- **Complexité** : pire cas `O(n^2)` comparaisons et déplacements.
- **Performance réelle** : efficace sur les petits tableaux ou tableaux déjà presque triés (cas meilleur en `O(n)` si déjà trié).
- Stable : l'ordre des éléments égaux est conservé.

Tri à bulles (Bubble sort)

Le **tri à bulles** répète des passages entiers sur le tableau, en échangeant les éléments adjacents mal ordonnés. On recommence jusqu'à ce qu'aucun échange ne soit nécessaire.

Pseudocode :
- Répéter :
- `swapped = false`
- Pour `i` de 0 à `n-2` :
- Si `array[i] > array[i+1]` :
- Échanger `array[i]` et `array[i+1]` ;
- `swapped = true`.
- Tant que `swapped` est vrai.

Implémentation en C :

```c
void tri_bulles(int *array, int n) {
    int swapped;
    do {
        swapped = 0;
        for (int i = 0; i < n - 1; i++) {
            if (array[i] > array[i+1]) {
                int temp = array[i];
                array[i] = array[i+1];
                array[i+1] = temp;
                swapped = 1;
            }
        }
        n--;  // optimisation : le dernier élément est déjà à sa place
    } while (swapped);
}
```

- **Complexité** : en moyenne et pire cas, `O(n^2)` comparaisons et échanges.
- **Performance réelle** : généralement très lente sur de grands tableaux. Bubble sort est souvent considéré comme un *anti-pattern* en pratique.
- Stable : comme insertion, l'ordre des éléments égaux est préservé.

Comparaison des algorithmes de tri

Voici un résumé comparatif des trois algorithmes simples de tri :

- **Complexité asymptotique** : tri par sélection, insertion et bubble sont tous en `O(n^2)` dans le pire cas.
- **Comparaisons et échanges** :
- *Selection sort* : ~ `n(n-1)/2` comparaisons et ~ `n-1` échanges fixes.
- *Insertion sort* : jusqu'à ~ `n(n-1)/2` comparaisons/déplacements dans le pire cas, mais peut être bien meilleur si le tableau est déjà partiellement trié.
- *Bubble sort* : ~ `n(n-1)/2` comparaisons et jusqu'à un nombre similaire d'échanges (beaucoup de swaps fréquents).
- **Performances réelles** : l'insertion sort est souvent plus rapide que bubble pour des tableaux petits ou partiellement triés. Bubble sort est le moins efficace en pratique (beaucoup de permutations inutiles). Selection sort fait peu d'échanges, mais effectue toujours le même nombre de comparaisons quoi qu'il arrive.
- **Stabilité** : insertion sort et bubble sort sont stables (ils ne changent pas l'ordre relatif des éléments égaux), contrairement au selection sort.

En résumé, malgré la même complexité en $O(n^2)$, **insertion sort** est souvent préféré aux deux autres pour les tableaux de taille modérée. Bubble sort, en raison de sa lenteur et de ses accès mémoire nombreux, est déconseillé sauf à des fins pédagogiques.

Fonction standard `qsort` et pointeurs de fonction

Le langage C fournit la fonction générique `qsort` (dans `<stdlib.h>`) pour trier n'importe quel tableau. Son prototype est :

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

- `base` : pointeur vers le début du tableau (généralement de type `void *`).
- `nmemb` : nombre d'éléments du tableau.
- `size` : taille en octets de chaque élément.
- `compar` : pointeur vers une fonction de comparaison.

Un **pointeur de fonction** est une variable qui contient l'adresse d'une fonction. Ici, `compar` est un pointeur vers une fonction qui prend deux pointeurs `const void *` (adressant deux éléments du tableau) et retourne un entier `<0`, `0` ou `>0` suivant que le premier élément est respectivement plus petit, égal ou plus grand que le second.

Exemple d'utilisation de `qsort` pour trier un tableau d'entiers :

```c
#include <stdio.h>
#include <stdlib.h>

// Fonction de comparaison pour int (ordre croissant)
int cmp_int(const void *a, const void *b) {
    int ai = *(const int*)a;
    int bi = *(const int*)b;
    if (ai < bi) return -1;
    if (ai > bi) return  1;
    return 0;
}

int main(void) {
    int array[] = {5, 3, 1, 4, 2};
    int n = sizeof(array) / sizeof(array[0]);
    // Appel à qsort : adresse du tableau, nombre d'éléments, taille d'un
élément, fonction de comparaison
    qsort(array, n, sizeof(int), cmp_int);
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
    return 0;
}
```

Ce programme affiche `1 2 3 4 5`. On voit que `qsort` utilise le pointeur `cmp_int` pour comparer les éléments.

Pour trier des structures ou d'autres types, on écrit simplement une autre fonction de comparaison adaptée. Par exemple, pour trier un tableau de structures de nombres complexes `(re, im)` par leur partie réelle :

```c
typedef struct { int re, im; } Complexe;

int cmp_complexe(const void *a, const void *b) {
    const Complexe *A = a;
    const Complexe *B = b;
    return (A->re - B->re); // tri selon la partie réelle
}
```

puis on appelle `qsort(table, n, sizeof(Complexe), cmp_complexe);`. Ainsi, `qsort` et les pointeurs de fonction permettent de réutiliser un même algorithme de tri pour différents types de données.
