Programmation C
TP n° 3 : Introduction aux pointeurs

Exercice 1 : Swap

1. Écrivez une fonction `void swap(int* pa, int* pb)` qui prend en argument deux pointeurs d'entiers et qui échange les valeurs aux adresses `pa` et `pb`.
2. Testez ensuite la fonction à l'aide des instructions suivantes :

```
int x,y;
x=5;
y=6;
printf(``Valeur de x avant échange : %d\n'',x);
printf(``Valeur de y avant échange : %d\n'',y);
swap(&x,&y);
printf(``Valeur de x après échange : %d\n'',x);
printf(``Valeur de y après échange : %d\n'',y);
```

3. Créez maintenant un tableau d'entiers avec au moins deux éléments et utilisez `swap` pour échanger les premières et dernières valeurs du tableau. Testez ce que vous avez fait de manière similaire à la question précédente.
4. On considère maintenant le bout de code suivant :

```
typedef struct{
  int x;
  int y;
} point;

point p={.x=5, .y=20};
```

   Écrivez des instructions qui échangent les valeurs des champs `x` et `y` de `p` grâce à la fonction `swap` et testez votre code.

Exercice 2 : Fonctions avec plusieurs paramètres de sortie

1. Écrivez une fonction `void minmax(int n, int t[], int *pmin, int *pmax)` qui donne les indices des plus petits et plus grand éléments du tableau `t` de taille `n`.
2. Écrivez une fonction `void occurences(int n, int t[], int e, int *pocc, int *first )` qui donne le nombre d'occurences de `e`, ainsi que l'indice de sa première occurence dans `t`.
3. Plutôt que de renvoyer l'indice de la première occurence, on peut renvoyer l'adresse de la case du tableau qui contient cette première occurence. Écrivez une fonction d'en-tête `void occurences_bis(int n, int t[], int e, int *pocc, int** first)` qui fait cela.

N'oubliez pas de tester toutes vos fonctions !

Exercice 3 : Le tri à bulles et ses variantes

Le tri à bulles (ou tri par propagation) est un algorithme de tri lent mais peu gourmand en mémoire. Son principe est le suivant :

1. On considère un tableau d'entiers t de taille n.
2. Pour i allant de 0 à $n-2$, on parcourt le tableau ; à chaque itération, si $t[i] > t[i+1]$, on les permute.
3. On applique cet algorithme au tableau t' constitué des $n-1$ éléments restants.

On constate en particulier qu'à la fin de l'étape (2), le maximum de t est placé à la fin de ce tableau, d'où la correction du traitement.

1. Écrivez une fonction `void sort(int t[], int start, int end)` qui trie les entiers stockés entre les indices `start` (inclus) et `end` (exclus). Pensez à utiliser la fonction codée à l'exercice 1.
2. Plutôt que de prendre en entrée des indices, on peut utiliser des pointeurs sur les cases correspondantes. Écrivez une fonction `void sort_point(int *start, int *end )` trie les entiers situés en mémoire entre les valeurs des pointeurs donnés en entrée. L'algorithme doit être le même qu'à la question précédente.
3. Définissez un tableau d'entiers, remplissez-le comme vous le souhaitez, et appliquez votre fonction entre deux adresses du tableau, puis affichez le tableau. Votre algorithme donne-t-il le résultat attendu ?

```
1       // Exemple :
2       int tab[] = {3,8,1,50,3,9,0,4,5,6,-7,9};
3       sort_bis(&tab[3],&tab[10]);
4       // Comment le contenu de tab a-t-il changé ?
```

   Une optimisation courante consiste à vérifier à chaque parcours entre les deux adresses mémoires si une permutation a bien eu lieu. Si ce n'est pas le cas, alors les éléments parcourus sont déjà triés et on peut mettre fin au traitement.
4. Écrivez une fonction `void opt_sort(int *start, int *end)` qui trie les entiers stockés entre les adresses des pointeurs `start` et `end` en implémentant l'optimisation précédente.