Programmation C
TP n° 3 : Introduction aux pointeurs

Exercice 1 : Opérations avec des pointeurs

Écrivez le programme fait de la suite d'instructions du tableau. À l'aide de `gdb`, posez un breakpoint (`break`) sur la ligne de l'instruction 2. Placez les variables dans la liste des affichages automatiques avec `display`. Essayez de prévoir la valeur des variables à chaque étape, puis vérifiez en lançant `next` (`n`).

	programme	`a`	`b`	`c`	`p1`, `*p1`	`p2`, `*p2`
1	`int a, b, c, *p1, *p2;`	×	×	×	×	×
2	`a = 1; b = 2; c = 3;`					
3	`p1 = &a, p2 = &c;`					
4	`*p1 = (*p2)++;`					
5	`p1 = p2;`					
6	`p2 = &b;`					
7	`*p1 -= *p2;`					
8	`++*p2;`					
9	`*p1 *= *p2;`					
10	`a = ++*p2 * *p1;`					
11	`p1 = &a;`					
12	`*p2 = *p1 /= *p2;`					

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'adresse de sa première occurence dans `t`.

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.

Dans cet exercice, nous allons implémenter le tri à bulles, mais en triant les entiers situés en mémoire entre deux valeurs de pointeurs. Le fonctionnement de l'algorithme doit être exactement le même qu'avec un tableau.

1. Écrivez une fonction `void swap(int *pa, int *pb)` qui échange les valeurs contenues aux adresses `pa` et `pb`.
2. Écrivez une fonction `void sort(int *start, int *end)` qui trie les entiers stockés entre les deux adresses mémoires `start` (inclus) et `end` (exclus).
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(&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.