Programmation C
TP n° 4 : Pointeurs (suite) et Allocation Dynamique

Exercice 1 : Pointeurs et tableaux

Que pouvez vous dire des bouts de codes suivants ?

1.

```c
int t[] = {1, 2, 3}, *pt;
pt = t;
```

```c
int t[3] = {1, 2, 3}, *pt;
pt = &t[0];
```

```c
int t[3] = {1, 2, 3}, *pt;
pt = t + 1;
```

```c
int t[3] = {1, 2, 3}, *pt;
pt = &t[1];
```

2.

```c
int t[] = {1, 2, 3}, *pt;
t = pt;
```

3.

```c
int t[3], *pt;
pt = malloc (5 * sizeof (int));
pt = t;
```

```c
int t[5] = malloc (5 * sizeof (int));
```

4.

```c
int *pt;
pt = malloc (5 * sizeof (int));
*pt = 10;
*(pt + 1) = 20;
*(pt + 12) = 30;
```

```c
int *pt;
pt = malloc (5 * sizeof (int));
pt[0] = 10;
pt[1] = 20;
pt[12] = 30;
```

Exercice 2 : Tableaux dynamiques

Dans cet exercice, nous allons utiliser des zones-mémoire allouées dynamiquement et manipulées à l'aide d'un unique type de structure.

```c
struct array {
  int *ptr;
  unsigned int capacite;
  unsigned int taille;
};
typedef struct array array;
```

Les valeurs de type `array` correctement initialisées seront appelées des *tableaux dynamiques* – même si ces "tableaux" ne sont pas à confondre avec les tableaux usuels.
L'idée est qu'on alloue initialement (par `malloc`) une zone de mémoire pouvant stocker au total `capacite` valeurs entières. Le champ `taille` (qui vaudra 0 au début) servira a compter le nombre de valeurs utiles actuellement dans notre tableau dynamique (quand on parlera des valeurs utiles d'un `array`, on parlera donc uniquement des valeurs stockees aux adresses comprises entre `ptr` et `ptr+(taille-1)`).

Bien sur, vous testerez dans le `main` vos fonctions dès que possible.

Remarque/Rappel : Lorsque l'on manipule un pointeur vers une structure, *e.g.* `array *p`, si l'on souhaite accéder au champ `size` d'une structure pointée par `p`, on pourrait écrire :

```
int n = (*p).taille;
```

Il existe un raccourci d'écriture, la "notation flèche" :

```
int n = p -> taille;
```

On vous impose dorénavant de toujours utiliser cette notation.

1. Écrire une fonction

```
array* array_init(unsigned int cap)
```

qui retourne l'adresse d'un array crée dynamiquement, de capacité donnée par `cap` et de taille 0. Attention, puisque on doit allouer l'espace pour un `struct array` ET pour la zone pointée par `ptr`, il y aura deux appels à `malloc` dans cette fonction. Si l'allocation échoue, on retournera `NULL`.

2. Écrire une fonction

```
void array_destroy(array *t)
```

qui libère la zone-mémoire allouée d'un tableau dynamique (il faut donc utiliser deux fois `free`, cf question précédente).

3. Écrire les fonctions

```
int array_get(array *t, unsigned int index);
void array_set(array *t, unsigned int index, int valeur);
```

permettant de lire et de modifier une valeur à un certain index dans un tableau dynamique d'adresse `t`. Il faut vérifier avec `assert` que l'indice passé en second argument est compris dans les indices des valeurs utiles.

4. Écrire la fonction

```
void array_append(array*t, int val)
```

qui ajoute une valeur à la fin du tableau, seulement si la capacité le permet.

5. Écrire une fonction

```
void array_print(array *t)
```

qui affiche les valeurs utiles du tableau dynamique d'adresse `t`, en séparant les valeurs par des espaces.

6. Écrire une fonction

```
int array_search(array *t, int val)
```

qui cherche la valeur `val` à l'intérieur du tableau : elle retourne l'indice de la première occurence si `val` est contenue dans le tableau, et -1 sinon.

7. Écrire une fonction

```
array* array_init_from(int* mem, unsigned int len, unsigned int cap)
```

qui retourne l'adresse d'un array crée dynamiquement, de capacite donnée par `cap` et dont les `len` premières valeurs sont obtenues en copiant les `len` premières valeurs de la zone mémoire pointée par `mem`. Cette fonction devra vérifier que `cap` est bien supérieure à `len` avec assert.

8. Écrire une fonction

```
void array_remove(array *t, unsigned int index)
```

qui supprime l'élément à la position `index`, avec un décalage vers la gauche de tous les éléments qui suivent l'élément supprimé.

9. Écrire une fonction

```
void array_insert(array *t, unsigned int index, int valeur)
```

qui permet d'insérer une valeur à un index donné dans un tableau dynamique d'adresse `t` en décalant les valeurs d'indice supérieure vers la droite.

La fonction n'autorisera pas un `index` d'insertion supérieur à la taille, mais elle acceptera qu'il lui soit égal : le nouvel element sera dans ce cas placé en dernière position. La fonction devra vérifier que la valeur d'`index` est correcte avec `assert`.

Attention : si la taille courante du tableau dynamique est égale à sa capacité avant l'ajout, il faudra d'abord réallouer une zone mémoire deux fois plus grande pour son espace de stockage (servez-vous de `realloc`).