Programmation C
TP n° 5 : Pointeurs (suite) et Allocation Dynamique

Les exercices indiqués avec une $*$ sont à rendre.

Exercice 1 : Pointeurs et tableaux

Que pouvez vous dire des bouts de codes suivants ?

1.

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

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

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

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

2.

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

3.

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

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

4.

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

```
int *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.

```
struct array {
  int *content;
  size_t capacity;
  size_t size;
};
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.

1. L'initialisation d'un tableau dynamique sera considérée comme correcte si son champ `content` est l'adresse de départ d'une zone-mémoire explicitement allouée par `malloc`, et si son champ `capacity` est la taille de de cette zone mémoire exprimée en nombre de valeurs `int` stockables à cette partir de cette adresse. La valeur de ce champ sera appelée *capacité* du tableau.
2. Le champ `size` sera initialisé à `0`. Il indiquera la *taille courante* d'un tableau dynamique, *i.e.* le nombre de valeurs que l'on considérera comme stockées dans ce tableau à un instant donné, entre les positions `0` et `size - 1`. Cette taille ne doit pas excéder la capacité du tableau.
3. Pour un tableau dynamique `a` donné, le *contenu courant* de `a` est la suite de valeurs `a.content[0]`, ... `a.content[td.size - 1]`.

Remarque. Lorsque l'on manipule un pointeur vers une structure, *e.g.* `array *pa`, si l'on souhaite accéder au champ `size` de la structure pointée par `pa`, on peut écrire :

```
size_t cs = (*pa).size;
```

On peut aussi utiliser le raccourci d'écriture suivant, la "notation flèche" :

```
size_t cs = pa -> size;
```

On vous impose dorénavant de toujours utiliser cette notation.

1. Écrire une fonction

```
array *array_init(size_t capacity)
```

qui retourne l'adresse d'un tableau dynamique alloué par `malloc`, de capacité `capacity` et de taille courante `0`. Attention, il faut deux mallocs : un pour la structure elle-même, un autre pour l'allocation de la zone d'adresse `content`. En cas d'échec d'une des deux allocations, la fonction retournera `NULL` (sans oublier de libérer la première zone allouée si la première allocation réussit, mais pas la seconde).

2. Écrire une fonction

```
void array_destroy(array *pa)
```

qui libère *toute* la zone-mémoire allouée pour un tableau dynamique d'adresse `pa` créé à l'aide de la fonction précédente (il faut donc deux `free`, dans le bon ordre).

3. Écrire les fonctions

```
int array_get(array *pa, size_t index);
void array_set(array *pa, size_t index, int value);
```

permettant respectivement de lire et de modifier la valeur à une certaine position dans un tableau dynamique d'adresse `pa`. Les deux fonctions devront vérifier avec `assert` que la position (`index`) est bien celle d'une valeur stockée dans le tableau (*i.e.* qu'elle est inférieure à sa taille courante).

4. Écrire la fonction

```
bool array_append(array *pa, int value)
```

Si la capacité du tableau dynamique d'adresse `pa` le permet, cette fonction doit ajouter au contenu courant du tableau la valeur `value` comme toute dernière valeur, puis renvoyer `true`. Sinon, la fonction renverra `false`.

5. Écrire une fonction

```
void array_print(array *pa)
```

qui affiche le contenu courant du tableau dynamique d'adresse `pa`, en séparant les valeurs par des espaces. Servez-vous de cette fonction pour tester chacune des précédentes, et des suivantes.

6. Écrire une fonction

```
int *array_search(array *pa, int value)
```

renvoyant : l'adresse de la première occurrence de `value` dans le contenu courant du tableau dynamique d'adresse `pa` si celle-ci existe ; `NULL` sinon.

7. Écrire une fonction

```
array *array_init_from(int *data, size_t length, size_t capacity)
```

Cette fonction devra vérifier avec `assert` que `capacity` est supérieur où égal à `length`. Elle doit renvoyer l'adresse d'un tableau dynamique alloué par `malloc`, de la capacité spécifiée, et dont la suite de valeurs sera une *copie* des `length` premières valeurs entières stockées à l'adresse `data`.

8. Écrire une fonction

```
void array_remove(array *pa, size_t index)
```

Cette fonction devra vérifier avec `assert` qu'`index` est bien la position d'une valeur stockée dans le tableau dynamique d'adresse `pa`. Elle doit supprimer cette valeur du contenu courant du tableau. Il faudra bien sûr décaler vers la gauche toutes les valeurs qui suivent celle supprimée.

9. Écrire une fonction

```
void array_insert(array *pa, size_t index, int value)
```

qui insère une valeur (`value`) à une position donné (`index`) dans un tableau dynamique d'adresse `pa` :

1. La fonction devra vérifier par `assert` que la position d'insertion n'est pas supérieure à la taille courante du tableau. Elle acceptera cependant qu'il lui soit égal : le nouvel element sera dans ce cas placé à la fin du contenu courant. Les valeurs de positions supérieures à la position d'insertion devront bien sûr être déclaées vers la droite.
2. Si la taille courante du tableau dynamique est égale à sa capacité avant l'insertion, la fonction commencera par réallouer une zone mémoire deux fois plus grande pour son espace de stockage `content` (servez-vous de `realloc`).