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

Exercice 1 : Pointeurs et tableaux

Que pouvez vous dire des bouts de codes suivants (erreurs de compilation ou d'exécution, commentaires) ?

1.

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

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

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

2.

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

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

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 : Durée de vie des variables locales

Recopiez le code ci-dessous dans un fichier `exo2.c`.

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

void affiche (int t[], int a, int b) {
  for (int i = a; i < b - 1; i++) {
    printf ("%d, ", t[i]);
  }
  printf("%d\n", t[b - 1]);
}
```

1. Que va-t-il se passer si l'on ajoute le main suivant ? Va-t-on avoir des erreurs de compilation ? d'execution ? Vérifiez votre réponse en ajoutant ce code.

```
int main () {
  int t[] = {1, 2, 3, 4};
  affiche (t, 0, 4);
  affiche (t, 0, 6);
  affiche (t, -2, 4);
  return 0;
}
```

2. Ajoutez après `affiche` la fonction suivante. Cette fonction est censée allouer un tableau de taille `n`, le remplir avec les valeurs `a`, `a + 1`,..., `a + n - 1`, puis renvoyer l'adresse de son premier élément. Elle n'est cependant pas correcte sous cette forme :

```
int *build_tab (int a, int n) {
  int t[n];
  for (int i = 0; i < n; i++) {
    t[i] = a + i;
  }
  printf  ("t : %p : ", t); // TEST
  affiche (t, 0, 15);       // TEST
  return t;                 // WARNING!
}
```

Pour vous en convaincre, remplacez le `main` précédent par celui ci-dessous. Compilez le code. Notez le message d'avertissement de `gcc`. Exécutez-le. Comment interprétez-vous l'affichage ? Décommentez l'appel d'`affiche` dans `main`. Que constatez-vous ?

```
int main (void) {
  printf ("\nbuild tab\n");
  int *tab = build_tab (0, 15);
  printf ("tab : %p\n", tab);
  //affiche (tab, 0, 15);    // SEG_FAULT!
  return 0;
}
```

3. Remplacez la ligne `int t[n]` par :

```
  int *t;
  t = malloc (n * sizeof (int));
```

et observez le résultat. Comprenez-vous ce qu'on entend par "durée de vie des variables locales" ? Quelle est la règle générale que l'on peut formuler sur ces variables ?

4. Vous avez vu en cours que `malloc` peut échouer à faire l'allocation et qu'il vaut mieux toujours être prudent lorsqu'on écrit une fonction qui retourne une adresse donnée par `malloc`. Proposez une solution pour que la fonction `build_tab` puisse tester la sortie de `malloc`.

Exercice 3 : 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
typedef struct array {
  int *ptr;
  int size;
} 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'initialisation d'une structure est considérée comme correcte si son champ `ptr` est l'adresse de départ d'une zone-mémoire explicitement allouée par `malloc`, et si son champ `size` est la taille de de cette zone mémoire exprimée en nombre de valeurs `int` stockables à cette partir de cette adresse.

Remarque. 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 peut écrire :

```c
int n = (*p).size;
```

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

```c
int n = p -> size;
```

Les fonctions suivantes pourront être testées avec le `main` ci-dessous – vous pouvez commencer par commenter les tests des fonctions non encore écrites.

1. Écrire une fonction `void array_print(array *t)` qui affiche le contenu du tableau dynamique d'adresse `t`, en séparant les valeurs par des espaces.

2. Écrire une fonction `array* array_init(int n)`. Cette fonction utilisera `malloc` pour allouer et retourner l'adresse d'un `array` de taille `n`. Attention, puisque on doit allouer l'espace pour un `array` ET pour le pointeur `ptr`, il y aura deux appels à `malloc` dans cette fonction. Si l'allocation échoue, on retournera `NULL`.

3. 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).

4. Écrire les fonctions `int array_get(array *t, int index)` et `void array_set(array *t, int index, int valeur)` permettant de lire et d'écrire une valeur à un certain index dans un tableau dynamique d'adresse `t`. Utilisez un `assert` pour imposer à `index` d'être dans les limites de sa zone-mémoire allouée. .

5. Écrire une fonction `int array_insert(array *t, int index, int valeur)`. Cette fonction doit permettre d'insérer une valeur à un index donné dans un tableau dynamique d'adresse `t`. La fonction devra :

1. Allouer une zone-mémoire de taille égale à celle du tableau dynamique, plus un. Si l'allocation échoue, elle doit retourner 0 ;
2. Recopier dans cette zone les valeurs du tableau dynamique d'index compris entre `0` et `index - 1`.
3. Écrire dans cette zone la valeur à la position `index`.
4. Recopier toutes les valeurs restantes du tableau dynamique d'index dans la zone allouée, immédiatement après la nouvelle valeur écrite.
5. Libérer la zone-mémoire du tableau dynamique initial.
6. Mettre à jour les champs `ptr` et `size` dans la structure.
7. retourner 1.

La fonction devra vérifier que la valeur d'`index` est correcte avec `assert` Elle acceptera que l'index d'insertion soit égal à la taille du tableau dynamique : le nouvel element sera dans ce cas placé en dernière position.

6. Écrire une fonction `void array_erase(array *t, int index)` effectuant l'inverse du traitement précédent. On codera cette fonction sans faire appel à `malloc`.

```c
int main() {
  array* t= array_init (10);
  array_print (t);   // 10 valeurs aleatoires, mais exactement 10.
  printf("\n");

  for (int i = 0; i < t->size; i++) {
      array_set (t, i, i);
  }
  array_print (t);
  printf("\n");
  // 0 1 2 3 4 5 6 7 8 9

  for (int i = 0; i < t->size; i++) {
      printf ("%d ", array_get (t, i));
  }
  printf ("\n");
  // 0 1 2 3 4 5 6 7 8 9

  array_insert (t, 3, 42);
  array_print (t);
  printf("\n");
  // 0 1 2 42 3 4 5 6 7 8 9

  array_insert (t, 11, 43);
  array_print (t);
  printf("\n");
  // 0 1 2 42 3 4 5 6 7 8 9 43

  array_erase (t, 11);
  array_print (t);
  printf("\n");
  // 0 1 2 42 3 4 5 6 7 8 9

  array_erase (t, 3);
  array_print (t);
  printf("\n");
  // 0 1 2 3 4 5 6 7 8 9

  array_erase (t, 0);
  array_print (t);
  printf("\n");
  // 1 2 3 4 5 6 7 8 9
  return 0;
}
```