Langage C
TP n° 5 : Zones de mémoire

Exercice 1 : Piles

Une pile (*stack* en anglais) est une structure de données dans laquelle les derniers éléments ajoutés sont les premiers à être récupérés. Une méthode pour implémenter une pile d'entiers consiste à stocker tous les éléments de la pile dans un tableau, où le dernier élément se trouve dans la dernière case remplie de ce tableau :

- lorsqu'on veut ajouter (empiler, *push* en anglais) un élément, on recopie tous les éléments (et le nouvel élément à la fin) dans un tableau plus grand d'une case ;
- lorsqu'on veut enlever (dépiler, *pop* en anglais) un élément, on recopie tous les éléments sauf le dernier dans un tableau plus petit d'une case.

Cette implémentation est simple mais inefficace puisque les éléments de la pile sont recopiés à chaque opération. Il est plus astucieux de garder un tableau partiellement rempli (méthode de la pile amortie) :

- lorsque la pile déborde, (c'est-à-dire lorqu'il n'y a plus assez de place pour empiler un nouvel élément) plutôt d'allouer un tableau plus grand d'une case, on alloue un tableau deux fois plus grand ;
- lorsque l'on dépile un élément, on ne recopie dans un tableau plus petit que si le tableau est aux trois quarts vide et de taille au moins deux, auquel cas on divise sa taille par deux.

On utilise la structure suivante pour mettre en œuvre la méthode de la pile amortie :

```c
typedef struct stack {
    int capacity;
    int length;
    point* buffer;
} stack;
```

où `capacity` est le nombre de cases du tableau sous-jacent à la pile, tandis que `length` représente le nombre de cases effectivement remplies. `buffer` est le tableau sous-jacent à la pile amortie.

En pratique on codera une pile dont les éléments sont des instances de la structure suivante, qui représente des points en 2 dimensions.

```c
typedef struct point {
    double x;
    double y;
} point;
```

On utilisera la fonction suivante pour afficher une pile :

```c
void stack_print(stack* s) {
    printf("Capacity: %d\n", s->capacity);
    printf("  Length: %d\n", s->length);
    for (int i = 0; i < s->length; i++) {
        printf("[%d] %20f %20f\n", i, s->buffer[i].x, s->buffer[i].y);
    }
}
```

Pensez à l'utiliser régulièrement pour tester vos fonctions.

1. On définit à l'aide d'une constante la capacité initiale `STACK_INITIAL_CAPACITY` d'une pile :

```
#define STACK_INITIAL_CAPACITY 1
```

Créez une fonction `stack *stack_new()` qui crée et alloue une nouvelle pile amortie de capacité initiale `STACK_INITIAL_CAPACITY`. On renverra `NULL` et on pensera à libérer toute mémoire éventuellement allouée si l'allocation échoue.

2. Créez une fonction `void stack_delete(stack *s)` qui libère la mémoire occupée par la pile `s` passée en paramètre. N'oubliez pas d'utiliser cette fonction pour libérer par la suite toute pile créée à l'aide de `stack_new`.

3. Écrivez une fonction `int stack_push(stack *s, point p)` qui empile le point `p` sur la pile `s`, et renvoie `0` en cas de succès et `-1` sinon. On veillera à utiliser `realloc` pour recréer le tableau si besoin.

4. Écrivez une fonction `int stack_pop(stack *s, point *p)` qui dépile un point de la pile `s` et le stocke à l'addresse indiquée par `p`. Cette fonction doit retourner `0` en cas de succès et `-1` sinon, par exemple si la pile est vide.

5. Écrivez une fonction `stack *stack_clone(stack *s)` qui copie la pile `s` passée en paramètre et retourne une pile indépendante qui contienne les mêmes éléments que `s` ou `NULL` en cas d'erreur. On utilisera `memmove` et non une boucle pour copier les éléments de la pile. Vous pourrez tester sur une pile qui n'est pas vide que copier la pile puis supprimer la pile d'origine laisse une seconde pile qui est la copie parfaite de la première pile.

6. Écrivez une fonction `int stack_push_vect(stack *s, int len, point *vect)` qui empile dans l'ordre sur la pile `s` les éléments d'un tableau `vect` de taille `len`. On utilisera `memmove` en non une boucle pour copier les éléments du vecteur sur la pile. Cette fonction renverra `0` en cas de succès et `-1` sinon.

7. Écrivez une fonction `int stack_pop_vect(stack *s, int len, point *vect)` qui dépile au plus `len` éléments de la pile `s` dans le tableau `vect`. Le tableau `vect` devra donc être au plus de taile `len`. La fonction renverra le nombre d'éléments dépilés ou `-1` en cas d'erreur. En effet si la taille de la pile est inférieure à `len` alors elle dépilera moins de `len` éléments. On utilisera `memmove` pour copier les éléments à dépiler dans le tableau et non une boucle.

La question suivante est optionelle, et doit être effectuée une fois le reste du TP fait.

8. Au lieu d'utiliser `#define STACK_INITIAL_CAPACITY 1` pour définir la capacité initiale de la pile on peut utiliser l'option de compilateur `-DSTACK_INITIAL_CAPACITY=1` pour spécifier à la compilation la capacité initiale. Enlevez la définition de `STACK_INITIAL_CAPACITY` donnée à la question 1 et spécifiez la capacité initiale directement à la compilation. Essayez de changer sa valeur à la compilation et observez les changements.

Exercice 2 : Les trois zones de la mémoire

Recopiez la macro suivante.

```
#define print_ptr(a) printf("%p (%lu) = %d\n", a, (unsigned long) a, *a)
```

Elle sert à afficher l'addresse d'un pointeur en hexadécimale (plus pratique pour repérer les puissances de 2 quand on a l'habitude) et en écriture décimale entre parenthèses (plus pratique pour faire des additions et des soustractions, ainsi que la valeur de la mémoire pointée par le pointeur après le `=`.

1. Créez deux variables globales de types `int` et affichez leur addresse à l'aide de `print_ptr` . Remarquez que les addresses se suivent en mémoire. On dit que ces variables sont stockées dans la *zone statique*.

2. Tout en gardant les appels à `print_ptr` de la question précédente, créez deux variables de types `int` dans la fonction `main` et affichez leur addresse à l'aide de `print_ptr`. Remarquez que les addresses se suivent en mémoire mais sont dans une zone différente que les variables de la question 1. On dit que ces variables sont stockées dans la *pile*.

3. Créez une fonction `void f(int n)` qui crée deux variables de types `int`, qui affiche leur addresse et qui appelle `f(n-1)` si `n` est strictement plus grand que `n`. (L'objectif de cette fonction est de simuler plusieurs appels de fonctions imbriqués et d'observer où les variables locales des fonctions sont stockées.) Appelez cette fonction avec exemple `n` égal 10 dans la fonction `main`. Remarquez que les variables se suivent approxivement en mémoire et que les variables d'un appel de fonction imbriqué se trouvent en mémoire avant les variables de la fonction qui la appelé. Observez ainsi que les variables allouées localement dans des fonctions le sont dans une zone mémoire qui se comporte comme une pile, d'où le nom de *pile* de cette zone mémoire.

4. Tout en gardant les appels des questions précédentes, allouez via `malloc` deux zones mémoires contenant chaqu'une un entier de type `int` et affichez les pointeurs retournés par ces appels à `malloc`. Remarquez que ces entiers sont alloués dans une nouvelle zone mémoire, que l'on nomme le *tas* (*heap* en anglais), et qui est réservée à la mémoire allouée via `malloc`.