Langage C
TP n° 12 : Pointeurs génériques et structures génériques

Exercice 1 :

1. Créez une fonction `void *my_memcpy(void *dest, void *src, size_t n)` qui copie `n` octets de la zone mémoire définie par `src` dans `dest`. On supposera qu'il n'y a pas chevauchement entre les zones mémoires `src` et `dest`.

2. Créez une fonction `void *my_memmove(void *dest, void *src, size_t n)` ayant un comportement identique à la question précédente, à l'exception que les zones mémoires définies par `src` et `dest` peuvent se chevaucher.

Exercice 2 : Piles génériques

Ce TP a pour but d'introduire les pointeurs générique en reprenant une structure de données déjà abordée dans le TP5.

Rappel : 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 égale à uen constante définie par `#define`, auquel cas on divise sa taille par deux.

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

```
1 typedef struct stack {
2     size_t capacity;
3     size_t length;
4     size_t te // la taille en octets d'un element de la liste
5     void* buffer;
6 } stack;
7
```

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, contenant des éléments de taille `te`.
Une représentation de pile d'entiers `int` et de pile de caractères `char` sont proposés sur Moodle (`stack_int_main.c` et `stack_char_main.c`). Votre pile générique devra fonctionner sur ces deux programmes.
Le fichier header `stack.h` est également fourni sur Moodle. L'implémentation de ces fonctions devra se faire dans un fichier `stack.c`.

1. Créez une fonction `stack *stack_init(stack *s, size_t te)` qui crée une pile amortie de capacité initiale égale à une constante définie par `#define`, chaque élément de taille `te`. Il faudra allouer dynamiquement la mémoire pour le tableau sous-jacent `buffer`. La pile sera stockée à l'adresse indiqué par `s`, et renvoyer cette adresse. La fonction retournera l'adresse `s` en cas de réussite, ou `NULL` en cas d'erreur (par exemple si l'allocation échoue).

2. Créez une fonction `void stack_cleanup(stack s)` qui libère la mémoire occupée par le tableau sous-jacent `buffer` défini dans la pile `s`. N'oubliez pas d'utiliser cette fonction pour libérer par la suite tout tableau de toute pile crée à l'aide de `stack_init`.

3. Écrivez une fonction `stack *stack_push(stack *s, void *el)` qui empile l'élément à l'adresse `el` sur la pile `s`. On veillera à utiliser `realloc` pour recréer le tableau si besoin. La fonction retournera l'adresse `s` en cas de réussite, ou `NULL` en cas d'erreur.

Remarque :
- pensez à utiliser la fonction `memmove` pour copier l'élément dans la pile
- parcourir un tableau à partir d'un pointeur générique `void *` peut s'avérer fastidieux. Vous pouvez créer une fonction auxiliaire privée `void *buffer_at(void * buffer, size_t te, size_t i)` qui renvoie l'adresse du i-ème élément de `s.buffer`, chaque élément étant de taille `te` octets.

4. Écrivez une fonction `stack *stack_pop(stack *s, void *el)` qui dépile un élément de la pile `s` et le stocke à l'adresse indiquée par `el`. La fonction retourne l'adresse `s` en cas de réussite, ou `NULL` en cas d'erreur.

5. Écrivez une fonction `stack *stack_clone(stack *s1, stack s2)` qui initialise une pile indépendante qui contienne les mêmes éléments que `s2`, et la stocke à l'adresse indiquée par `s1`. On utilisera `memmove` et non une boucle pour copier les éléments de la pile. La fonction retournera l'adresse `s1` en cas de réussite, ou `NULL` en cas d'erreur.

6. Écrivez une fonction `stack *stack_push_vect(stack *s, size_t len, void *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. La fonction retournera l'adresse `s` en cas de réussite, ou `NULL` en cas d'erreur.

7. Écrivez une fonction `stack *stack_pop_vect(stack *s, size_t *slen, size_t len, void *vect)` qui dépile au plus `len` éléments de la pile `s` dans le tableau `vect`, de taille au plus `len`. Si la taille de la pile est inférieure à `len` alors elle dépilera moins de `len` éléments. Le nombre d'éléments dépilés sera stocké à l'adresse `slen` si elle ne vaut pas `NULL`. On utilisera `memmove` pour copier les éléments à dépiler dans le tableau et non une boucle. La fonction retournera l'adresse `s` en cas de réussite, ou `NULL` en cas d'erreur.

Exercice 3 : Initiation aux pointeurs de fonctions

Actuellement dans le code des programmes, l'affichage d'une pile se fait via la fonction `void stack_int_print(stack s)` et `void stack_char_print(stack s)`.
Cette fonction n'est pas générique car elle prend en considération le type de l'élément. Mais il est possible de généraliser le reste du code de cette fonction si l'on sait au moins comment afficher un élément.
Dans la structure de la pile, nous allons rajouter le champs `void (*print_element)(void *el)` représentant la fonction d'affichage d'un élément.

```
typedef struct {
    size_t capacity;
    size_t length;
    size_t te // la taille en octets d'un element de la liste
    void* buffer;
    void (*print_element)(void *el);
}

```

1. Modifiez la fonction `stack *stack_init(stack *s, size_t te, void (*print_element)(void *el))` pour créer une pile amortie dont chaque élément s'affiche avec `print_element`.

2. Créez une fonction `void stack_print(stack s)` qui affiche les informations de la pile `s`, puis chaque élément du tableau sous-jacent avec `s.print_element`. (pensez à rajouter sa définition dans le header `stack.h`)

3. Dans les programmes `stack_int_main.c` et `stack_char_main.c`, créez respectivement les fonctions `void print_int(void *el)` et `void print_char(void *el)` qui affiche la valeur de l'élément à l'adresse `el` en tant qu'entier `int` ou caractère `char`.

Vous pouvez enfin tester l'affichage d'une pile (par exemple dans `stack_int_main.c`) :

```
void print_int(void *el) {
  ...
}

int main() {
    stack s;
    stack_init(&s, sizeof(int), print_int);
    stack_push(&s, 1);
    stack_push(&s, 2);
    stack_push(&s, 3);
    stack_print(s);
    stack_cleanup(s);
    ...
}

```