Langage C
TP n° 5 : Zones de memoire

Exercice 1 : Les trois zones de la mémoire

Recopier la macro suivante.

```
1 #define affiche_nbr(a) printf("%p (%lu)\n", a, (unsigned long) a)
```

Elle sert à afficher un nombre entier en écriture 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).
Pour afficher l'adresse d'une variable `n`, vous utiliserez l'instruction `affiche_nbr(&n)`, pour afficher l'adresse contenue dans une variable `p` de type pointeur, vous utiliserez `affiche_nbr (p)` (et bien sûr `affiche_nbr(&p)` pour afficher l'adresse de la variable `p`.)

La pile (*stack*) et la zone statique

1. Déclarez et initialisez les variables suivantes :
- deux variables globales (c'est-à-dire en dehors de toute fonction) de type caractère ;
- deux variables globales de type entier ;
- deux variables globales de type pointeur (`int *` par exemple) ;
- deux variables de type caractère dans la fonction `main` ;
- deux variables de type entier dans la fonction `main` ;
- deux variables de type pointeur d'entier dans la fonction `main`.

Affichez les adresses de chacune de ces variables. Vous remarquerez que :
- les variables globales ne sont pas stockées au même endroit que les variables définies dans une fonction (pour les premières, on l'appelle la zone statique, pour les dernières, la pile).
- les variables se suivent dans la mémoire, mais pas dans le même ordre dans les deux zones de la mémoire.

2. Affichez les valeurs de `sizeof(int)`, `sizeof(char)` et `sizeof(int *)` et comparez avec le nombre d'octets qui séparent les variables déclarées à la question précédente. Vous remarquerez que ce nombre d'octets est parfois supérieur (le processeur manipule les octets par blocs de 4, voire de 8, donc il « arrondit » les adresses pour se simplifier la tâche, on dit qu'il les *aligne*).

3. Écrivez une fonction `void f1(int *p)` dans laquelle vous déclarerez une variable de type `int`, initialisée à 7 et une de type `char`, initialisée à `'A'`. Dans cette fonction, vous afficherez :
- le contenu de l'adresse pointée par le paramètre `p`
- la valeur de `p`
- l'adresse de `p`
- l'adresse et le contenu de chacune des variables

Après l'affichage, vous modifierez la valeur de l'entier pointé par `p` *puis* la valeur de `p` (en lui affectant la valeur `NULL`).

4. Dans la fonction `main`, affectez à l'un des pointeurs d'entiers l'adresse d'une des variables entières déclarées à la question 1. Appelez la fonction `f1` dans le `main` en lui passant

comme argument ce pointeur. Après l'appel, affichez l'adresse et la valeur de ce pointeur, ainsi que la valeur de l'entier pointé. L'appel à la fonction a modifié l'entier mais pas le pointeur (lors de l'appel, le contenu du pointeur a été recopié, et continue de pointer sur le même endroit de la mémoire).

5. Écrivez une fonction `void f2(int *p)` dans laquelle vous déclarerez deux variables de type `int` non initialisées. Dans cette fonction, vous afficherez :

- l'adresse du paramètre ;
- l'adresse de chacune des variables ;
- le contenu des variables.

Dans la fonction `main`, appelez cette fonction *juste* après avoir appelé `f1`. Vous remarquez que les adresses des paramètres et des variables sont les mêmes que lors de l'appel de `f1` : La place mémoire utilisée par `f1` a été libérée quand la fonction s'est terminée, donc elle est reprise à l'identique pour l'appel à `f2`. De ce fait, les variables non initialisées de `f2` contiennent ce qui avait été mis là par `f1`. Comparez aussi les adresses des variables locales déclarées dans `f1` et `f2`.

6. Dans la fonction `f2`, appelez la fonction `f1`. Vous constatez que les adresses des variables et des paramètres sont différentes : la zone mémoire utilisée par `f2` n'a pas encore été libérée.

Le tas (*heap*)

7. Dans le `main`, affichez les adresses renvoyées par cinq appels successifs à la fonction `malloc`, pour réserver successivement 1, 2, 25, 26 puis 1 octet de mémoire.

Les adresses allouées se trouvent dans une nouvelle zones de mémoire appelée le tas. Par ailleurs, vous remarquerez que le nombre d'octets qui sépare deux zones allouées est plus grand que la taille de la zone allouée. La place supplémentaire est occupée par des informations stockées par `malloc`, qui serviront notamment à la fonction `free`.

Exercice 2 : Piles

Une pile 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 (le sommet de la pile se trouve dans la dernière case remplie de ce tableau) :

- lorsqu'on veut ajouter (empiler) 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 le dernier élément (dépiler), 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, plutôt que 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, auquel cas on divise sa taille par deux.

Conseil (pour aller plus vite) : on pourra reprendre/s'inspirer d'une grande partie du code source du TP4 (tableaux dynamiques). Notez cependant que, dans cet exercice, il n'était pas demandé de gérer la possible diminuation de la taille du tableau.

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

```
typedef struct pile_amortie {
  int occupation;
  int capacite;
  int *elements;
} pile_amortie;
```

où `capacite` représente le nombre de cases du tableau `elements`, tandis que `occupation` représente le nombre de cases effectivement remplies. À partir de la case numéro `occupation` il y a des cases non utilisées.

1. Écrire une fonction `pile_amortie *alloue_pile_amortie()` qui crée une pile amortie de capacité initiale 0.

2. Écrire une fonction `void libere_pile_amortie(pile_amortie *pile)` qui libère la mémoire occupée par la pile `pile`. On veillera à bien libérer toute la mémoire occupée.

3. Écrire une fonction `void empile_pile_amortie(pile_amortie *pile, int n)` qui empile l'entier `n` sur la pile `pile`.

4. Écrire une fonction `int depile_pile_amortie(pile_amortie *pile)` qui dépile la pile `pile` et renvoie l'élément dépilé.