DM #2 — Petit TP des vacances d'automne

Exercice 1. *Allocation, désallocation & initialisation*

1. Écrivez une fonction C *alloc* qui prend un paramètre entier n, alloue une zone mémoire de durée de stockage « allouée » capable de contenir n `int`s, et renvoie un pointeur vers le début de cette zone.
2. Écrivez une fonction C *dealloc* qui permet de libérer une zone mémoire créée par *alloc*.
3. Écrivez une fonction C de signature :
   ```
   void init(size_t n, int *p)
   ```
   dont l'argument p doit pointer vers une zone mémoire capable de stocker n valeurs de type `int`, et qui initialise chacune de ces valeurs à `0`.
4. Testez chacune de vos fonctions. Dans le cas de la fonction *init*, celle-ci devra au moins être testée deux fois : l'une avec un second argument égal à un tableau, l'autre avec un second argument égal à un pointeur renvoyé par *alloc*.

   Lors de ces tests, **aucune** erreur ne doit être détectée par l'*address sanitizer*.

Exercice 2. *All teh errors*

1. Écrivez de petits programmes (d'une dizaine de lignes maximum) permettant de déclencher (au moins une fois) chacune des erreurs suivantes telles que reportées par l'*address sanitizer* (dans sa version pour *clang* ; il n'est pas important que l'intitulé de l'erreur soit *littéralement* le même) :

   — `SEGV`
   — `stack-buffer-overflow`
   — `heap-buffer-overflow`
   — `stack-use-after-return`
   — `heap-use-after-free`
   — `stack-use-after-scope`
   — `attempting double-free`
   — `attempting free on address which was not malloc()-ed`
   — `detected memory leaks`

2. Que pouvez-vous constater par rapport aux avertissements émis par le compilateur ?

Exercice 3. `memcmp`

1. Écrivez une fonction C *mymemcmp* de signature :
   `int mymemcmp(size_t n, uint8_t *a, uint8_t *n)`
   qui prend en entrée deux arguments a et b pointant chacun vers une zone mémoire contenant au moins n octets, et renvoie :
   - `0` si les n premiers octets de ces zones mémoires sont égaux, quand interprétés comme des entiers non signés de 8 bits.
   - `1` s'ils sont différents et que le premier octet différent entre a et b est plus grand dans a que dans b (quand interprété comme un entier).
   - `-1` dans tous les autres cas.
2. Testez *via* une fonction de test sur des cas variés. *Aucune* erreur ne doit être déclenchée par l'*address sanitizer* lors de ces tests.

Exercice 4. `memmove`

1. Écrivez une fonction C *mymemmove* de signature :
   `void mymemmove(size_t n, uint8_t *src, uint8_t *dst)`
   qui prend en entrée deux arguments *src* et *dst* pointant chacun vers une zone mémoire contenant au moins n octets, et copie les n octets de la zone pointée par *src* vers la zone pointée par *dst* (dans le même ordre).
   Attention : vous ne pouvez pas supposer que les zones pointées par *src* et *dst* sont disjointes. Autrement dit, elles peuvent se « chevaucher ».
   Attention : votre fonction doit pouvoir s'exécuter correctement dans un environnement usuel quand n est « grand » (par exemple égal à 2^{30}).
2. Testez *via* une fonction de test sur des cas variés. *Aucune* erreur ne doit être déclenchée par l'*address sanitizer* lors de ces tests.

Exercice 5. *Tri par comptage*

On souhaite généraliser le tri « chèvres et moutons » du TP#4 à un nombre d'éléments supérieur à deux : soit a un tableau de n entiers de type `unsigned` et inférieurs ou égaux à un certain entier k, on souhaite renvoyer un pointeur vers une zone mémoire de durée de stockage « allouée » contenant n entiers de type `unsigned`, qui contienne les mêmes éléments que a (avec multiplicité) triés par ordre croissant.
Pour cela, on va utiliser l'algorithme de *tri par comptage* suivant :

- Dans un premier temps, on dresse un histogramme des valeurs présentes dans a, c'est à dire que pour chaque entier $x \in [\![0, k]\!]$ on compte combien de fois il apparaît dans a.
- Dans un second temps, on calcule pour chaque entier $x \in [\![0, k]\!]$ la valeur x_i du plus petit indice tel que l'on trouve une valeur strictement supérieure à x dans un tableau trié de mêmes éléments que a. Autrement dit, dans un tel tableau les éléments égaux à x se trouvent exactement aux indices $\in [\![x_{i-1}, x_i - 1]\!]$ (en prenant $x_{-1} = 0$.

— Enfin, on construit une solution en écrivant le bon nombre d'éléments au bon endroit.

1. Écrivez une fonction C de signature :
   `unsigned *count_sort(size_t n, unsigned a[n], unsigned k)`
   qui implémente cet algorithme.
2. Testez *via* une fonction de test sur des cas variés.