Théorie

Comment utiliser les fonctions ?

Un simple appel de `my_malloc(taille)` pour allouer de la mémoire dans le tas et un simple appel de `my_free(pointeur)` pour libérer la mémoire allouée par le `my_malloc`.

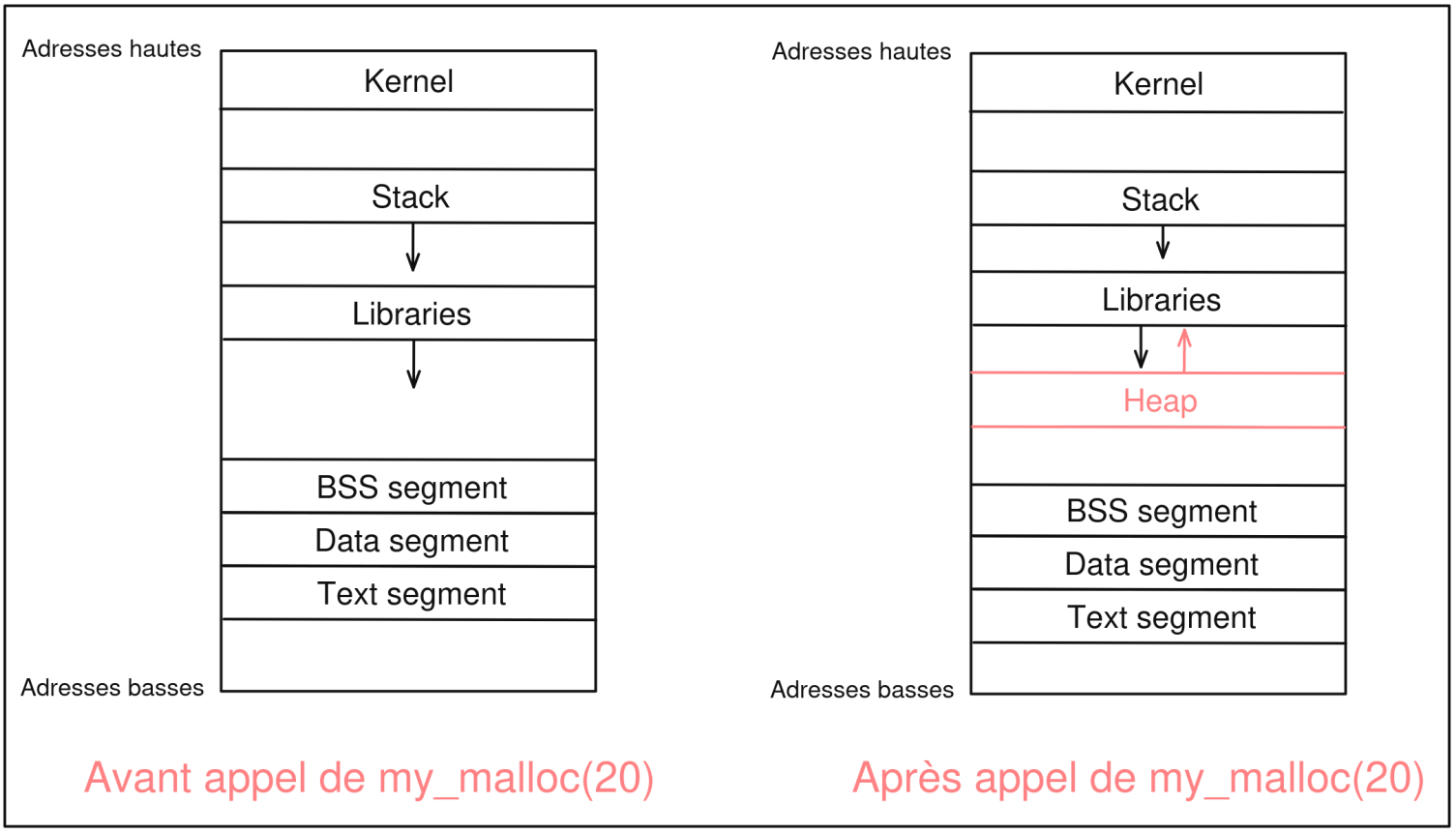
Tu veux allouer un bloc mémoire dans le tas où chaque octet est initialisé à 0 ? Tu peux utiliser la fonction `my_calloc(A,B)` qui va allouer de la mémoire pour un tableau de `A` éléments, dont chaque élément a une taille `B`. Exemple : Si je veux allouer de la mémoire pour un tableau de 80 entiers (codés sur 4 octets) où chaque entier est initialisé à 0 j'appellerai `my_calloc(80,4)`.

Tu souhaites modifier la taille d'un bloc mémoire déjà alloué par `my_malloc` ? Tu peux utiliser la fonction `my_realloc(A,B)` où `A` représente le pointeur vers le bloc mémoire déjà alloué et `B` la nouvelle taille de ce bloc mémoire. Ainsi, `my_realloc` renverra un pointeur vers le nouveau bloc mémoire. Exemple : `my_realloc(my_alloc(10),99)` permet de modifier la taille d'allocation de 10 octets théoriques à 99 octets théoriques.

Mais comment fonctionne la heap et l'appel de `my_malloc` dans la heap ? Reprenons depuis le début.

Heap et top_chunk

La **heap**, c'est l'endroit qui te permet d'allouer de la mémoire de manière dynamique : tant que tu n'alloues pas des éléments dans le tas, le tas n'existe pas !

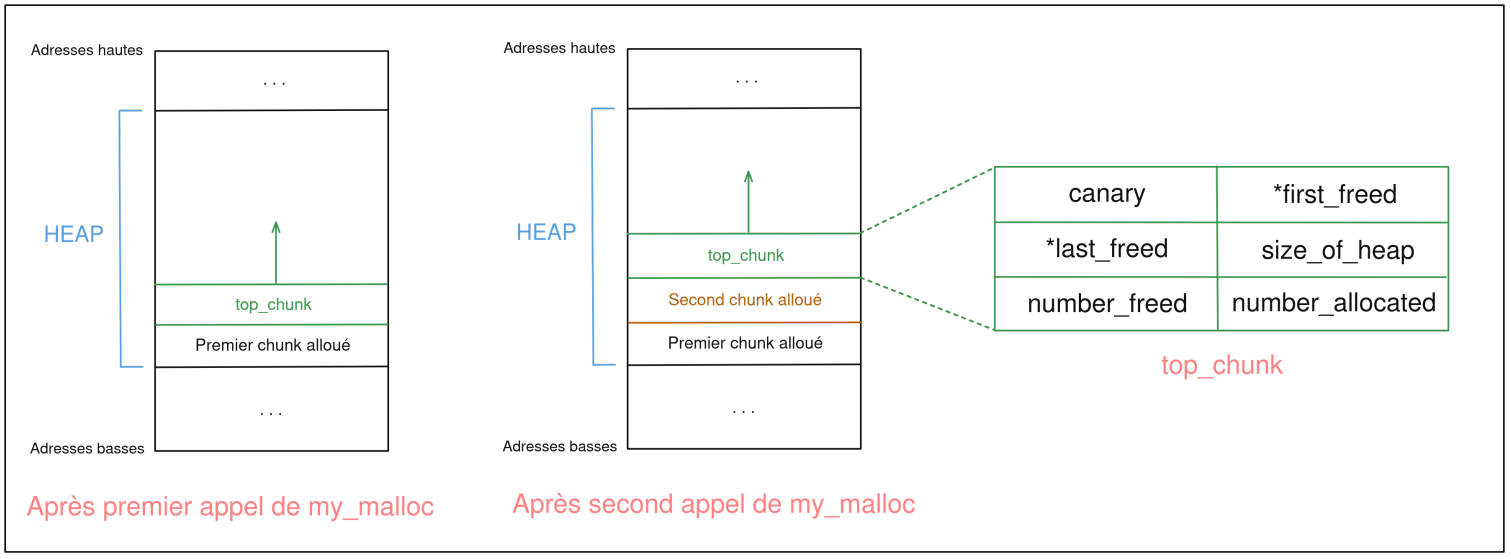


Cette **heap** se forme grâce à un premier élément important : le **top_chunk**. C'est l'élément qui se trouvera **TOUJOURS** vers l'adresse la plus haute de la **heap**, et qui va dicter tout ce qu'il se passe dedans. Voici sa structure :

```c
struct top_chunk{
    // 6 champs de 8 octets <=> 6*8 = 48 octets donc forcément aligné sur 0x10 octets (16 octets)
    // On aligne pour se simplifier la vie quand on alloue et libère de la mémoire dans la heap
    size_t canary;
    size_t *first_freed;
    size_t *last_freed;
    size_t size_of_heap;
    size_t number_freed;
    size_t number_allocated;
}
```

- **canary** : Valeur aléatoire qui sera utilisée pour implémenter la sécurité. *Pour le moment on s'en fiche.*
- **first_freed** : Pointeur pointant vers le premier élément qui a été free dans la liste chaînée. *On s'en fiche aussi.*
- **last_freed** : Pointeur pointant vers le dernier élément qui a été free dans la liste chaînée. *On s'en fiche également.*
- **size_of_heap** : Valeur représentant la taille totale de la **heap**. Cette valeur est importante car elle permet de déterminer jusqu'où s'arrête la **heap**, c'est-à-dire jusqu'où on peut allouer de la mémoire. Si la totalité de la **heap** est allouée et qu'on souhaite allouer + de mémoire, deux solutions s'offrent à nous : Soit il y a encore de l'espace dans la mémoire virtuelle du processus et on peut agrandir la taille de la **heap**, soit il n'y a plus d'espace et il est impossible d'allouer + de mémoire donc on quitte le programme.
- **number_freed** : Valeur représentant le nombre total de blocs mémoire libérés dans la **heap**. Cette valeur nous permettra d'implémenter la liste chaînée des éléments qui ont été free, en utilisant les pointeurs `first_freed` et `last_freed`.
- **number_allocated** : Valeur représentant le nombre total de blocs mémoire alloués dans la **heap**. Cette valeur permettra de savoir si l'utilisateur a déjà alloué de la mémoire (au moins un appel à `my_malloc` ou `my_realloc`) afin d'y insérer le **top_chunk**. Car oui, si la **heap** n'existe pas alors le **top_chunk** non plus et la réciproque est vraie. Mais si la **heap** existe, c'est que l'utilisateur a fait **AU MOINS** un appel à `my_malloc` ou `my_realloc`.

Voici un schéma pour mieux comprendre :



Ce sera à nous de définir la taille de départ de la **heap**, car comme tu l'auras compris, quand tu alloues de la mémoire pour la première fois `top_chunk->size_of_heap` est initialisé à ce que l'on aura défini, et quand tu alloues plusieurs fois de la mémoire `top_chunk->size_of_heap` diminuera au fil du temps.

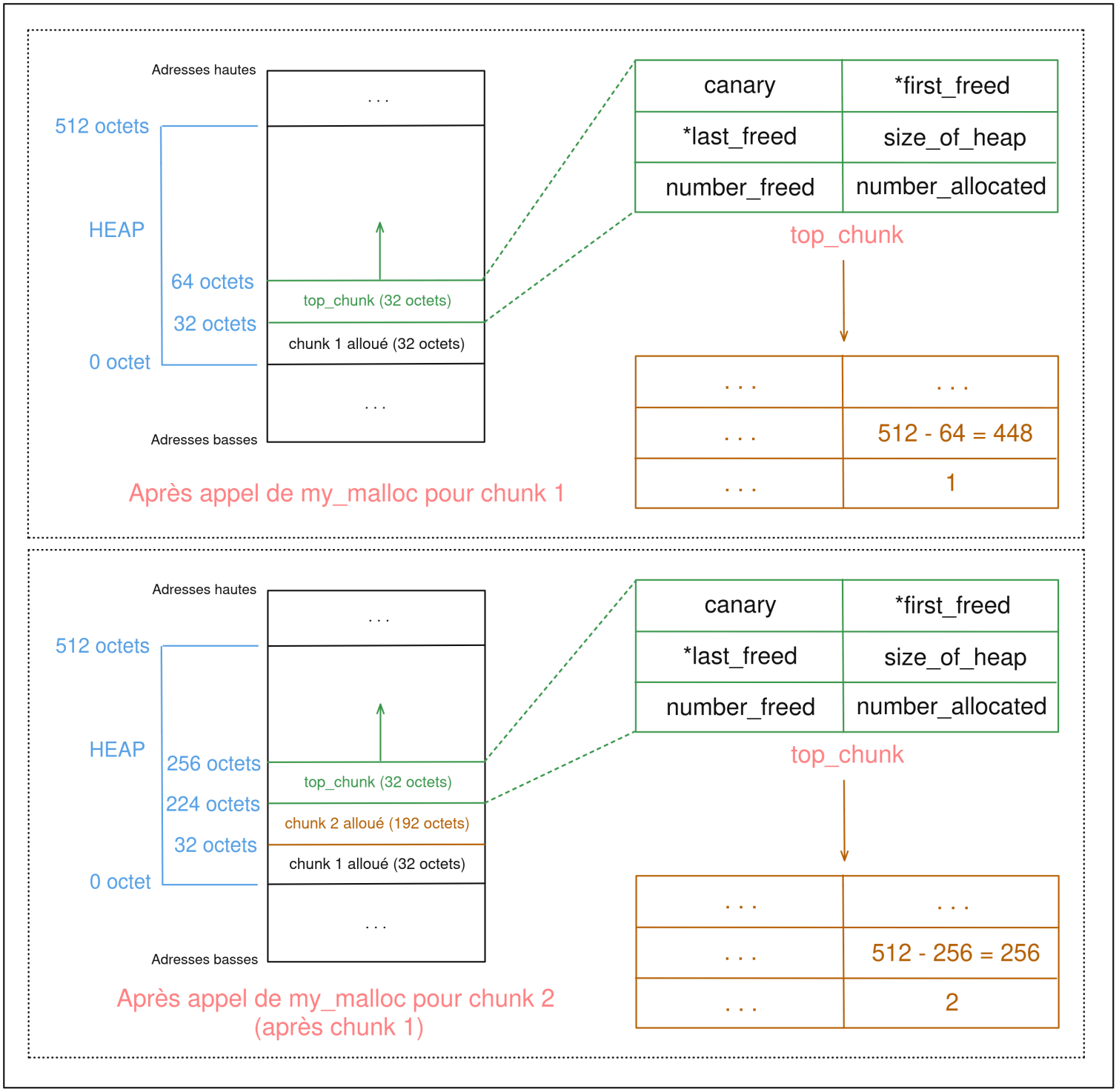
On pourra utiliser `heap` comme variable globale et `top_chunk->number_allocated` :

- Si `heap` vaut **NULL** alors l'utilisateur n'a pas encore fait d'allocation, il faudra créer une fonction pour initialiser la **heap** et le **top_chunk**.
  - Initialiser `heap` en global (accessible pour tout le monde) par le pointeur que retournera `mmap` de taille que l'on choisira.
  - Initialiser `top_chunk` comme ceci :
    - `canary` = valeur aléatoire de */dev/urandom*
    - `first_freed` = NULL
    - `last_freed` = NULL
    - `size_of_heap` = taille à choisir
    - number_freed = 0

    - number_allocated = 1
- Si `heap` n'est pas **NULL**, alors le `top_chunk` existe.
    - Si `top_chunk->number_allocated == 0` alors on s'apprête à quitter le programme :
        - en utilisant le syscall `munmap` nous permettant d'unmapper des zones mémoires (qui ont été mappées avec `mmap`). Il suffit d'unmapper l'entiereté de la **heap**.
    - Si `top_chunk->number_allocated == 1` alors le **top_chunk** est présent dans la **heap** mais aucune allocation n'est encore effectuée.
    - Si `top_chunk->number_allocated > 1` alors le **top_chunk** est présent dans la **heap** et au mois une allocation est faite.

ATTENTION : en utilisant ce système, le nombre de blocs mémoires réellement alloués dans la heap est `(top_chunk->number_allocated - 1)` car il faut enlever le top_chunk qui correspond simplement à de la métadonnée.

Voici un exemple sur les champs `top_chunk->size_of_heap` et `top_chunk->number_allocated` :



Ok, c'est super, mais depuis tout à l'heure on parle de **chunk**. C'est quoi ?

Chunk

Le **chunk** représente le bloc mémoire qui a été alloué et qui contient de la donnée que l'utilisateur va pouvoir lire / écrire. Il a à peu près la même gueule que le **top_chunk** à quelques exceptions près. Contrairement au **top_chunk**, le **chunk** n'est pas considéré comme de la métadonnée mais comme de la véritable donnée.

Voici sa structure :

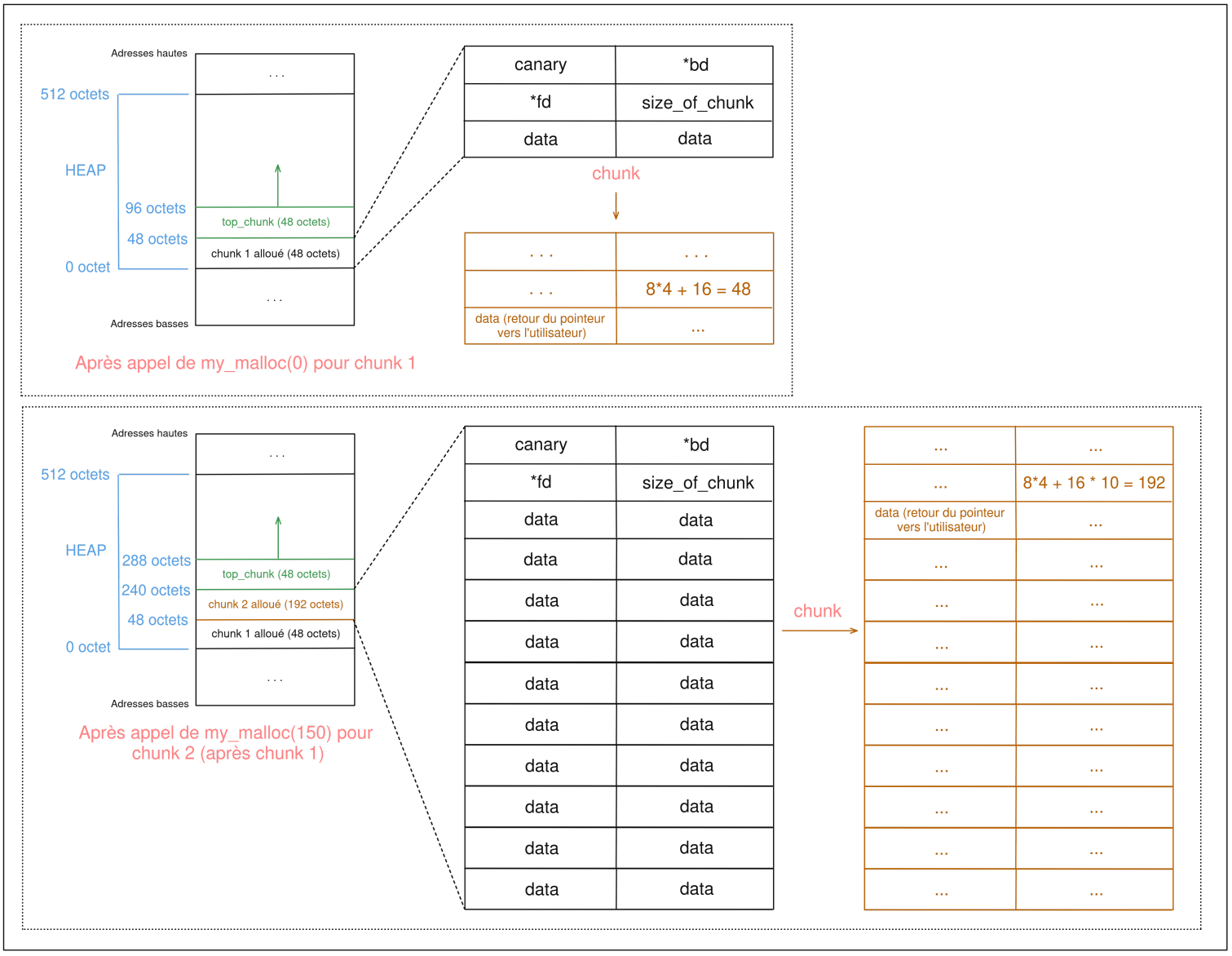
```
struct chunk{
    // 5 champs avec les 4 premiers champs à 8 octets et le 5ème qui est aligné sur 16 octets
    // En considérant n la taille quelconque (entière et supérieure ou égale à 1), on a
    // (8*4 + n*16) = (2*16 + n*16) = (16 * (2+n)) donc on a bien un alignement sur 16 octets
    size_t canary;
    size_t *bd;
    size_t *fd;
    size_t size_of_chunk;
    size_t *data;
}
```

- **canary** : Pareil que pour le **top_chunk**, valeur aléatoire dans */dev/urandom* implémentant de la sécurité (on verra plus tard)
- **bd** : Le *Backward pointer* est le pointeur pointant vers l'élément qui a été free et qui pointe vers une adresse plus grande (on verra plus tard)
- **fd** : Le Forward pointer est le pointeur pointant vers l'élément qui a été free et qui pointe vers une adresse plus petite (on verra plus tard)
- **size_of_chunk** : Taille totale du **chunk** en octets.
- **data** : Pointeur pointant vers la zone mémoire qui sera lue / écrite par l'utilisateur. C'est ce pointeur qui est retourné à l'utilisateur. Ce champs a une taille minimale de 16 octets et est aligné sur 16 octets. Pour calculer sa taille :
  - size_of_data = size_of_chunk - 4 * size_t

Comment implémenter la fonction `my_alloc(taille)` (avec une taille entière et supérieure ou égale à 0) avec les problèmes d'alignement sur 16 octets concernant **data** ?

- Si `taille` est égale à 0, la taille de **data** est de 16 octets (minimal par défaut).
- Si `taille` est égale à un nombre divisible par 16, la taille de **data** vaut ce nombre.
- Si la `taille` est égale à un nombre **non** divisible par 16, la taille de **data** vaut le nombre divisible par 16 qui est supérieur à `taille` et qui en est le plus proche. En d'autres termes, `data_size = ((int)(taille / 16) +1) * 16
  - `my_alloc(25)` revient à dire que **data** a une taille de 32 octets car `ceil(25 / 16) * 16 = ceil(1.5625) * 16 = 2 * 16 = 32`

Un schéma vaut mille mots :



C'est bon, on a les bases pour pouvoir allouer de la donnée ! Et comment on la libère cette donnée ? Avec `my_free`.