L'objectif

Le but de ce TD est de se familiariser avec :

- l'allocation dynamique de mémoire.
- l'arithmétique des pointeurs.
- le traitement des pointeurs en général (pointer cast, ...).

Vous devrez implémenter une structure de données appelée tableau dynamique. Cette structure de données est similaire à l'`ArrayList` en Java ou au `std::vector` en C++.
Un tableau dynamique est un tableau dont la taille change à chaque fois que nous y ajoutons un élément, contrairement aux tableaux statiques typiques que nous avons vus jusqu'à présent en C. Un tableau dynamique a de nombreuses implémentations différentes et une liste chaînée peut en faire partie. Cependant, dans ce TD, nous implémenterons l'implémentation la plus courante (celle qui est utilisée dans `std::vector` en C++). Dans cette implémentation, les éléments sont stockés de manière contiguë dans la mémoire comme les tableaux C classiques.

Détails de l'implémentation

Notre tableau dynamique va être présenté par une structure appelée `vector`[1].
La structure aura les champs suivants:

```
1 typedef struct vector_t
2 {
3     void*   elements;
4     size_t  size;
5     size_t  capacity;
6     size_t  unit_size;
7 } vector;
```

elements : Comme mentionné dans la section précédente, les éléments du tableau doivent être contigus en mémoire. `elements` est donc un pointeur vers une région de mémoire qui contiendra les éléments de notre tableau dynamique.
size : Représente le nombre d'éléments actuellement stocké dans notre tableau dynamique.
capacity : Représente la capacité totale que la mémoire allouée peut gérer. En d'autres termes, la capacité représente le nombre d'éléments que notre tableau peut contenir avant d'être considéré comme plein. Cela deviendra crucial plus tard pour obtenir une complexité d'insertion qui est de l'ordre de $O(1)$ amortie.
unit_size : représente la taille de chaque élément en octets. Ce champ est important car nous voulons que notre tableau contienne n'importe quel élément tant qu'ils sont du même type.

Exemple : Dans l'image ci-dessous, une représentation d'un tableau dynamique (`vector`) contenant le type fondamental `int` est donnée. Notez que le tableau pointé par le champ `elements` peut contenir 8 éléments (ce chiffre est représentés par le champ capacité). Notez également que le tableau ne contient que 5 éléments (représenté par `size`) et que le reste de la mémoire est considéré comme vide. La taille de la mémoire allouée en octets est $capcity * unit\_size$.

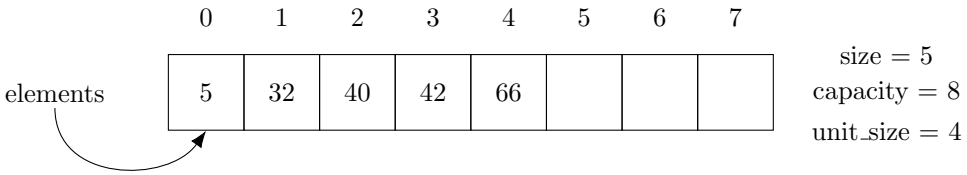

Figure 1 - Un exemple d'un `vector` de `int`

[1] les autres noms possibles sont `DynamicArray`, `ArrayList`, ...

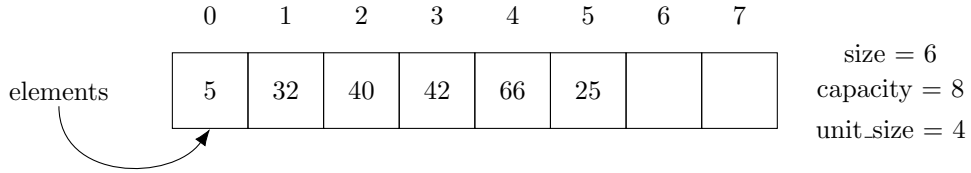

Figure 2 - Ajout de l'élément 25. Aucune allocation n'a eu lieu

Si on ajoute l'entier 25 le tableau sera présenté par la figure ci-dessus. Remarquez qu'on plaçe l'élément en mémoire et incrémente le champ `size` sans allouer de mémoire puisque il y a encore de la mémoire vide autrement dit `size` est encore inférieur à `capacity`.

Imaginons maintenant qu'on ajoute les entiers 50, 60, 70 à notre `vector`. Dans ce cas, la capacité va être dépasser donc on doit réallouer de la mémoire mais au lieu d'allouer $(capacity + 1) * unitSize$ octets on va allouer $(capacity * 2) * unitSize$ octets tout en copiant les éléments déja présents de l'ancien zone mémoire vers la nouvelle zone mémoire. Donc plus tard, on n'a pas à allouer de la mémoire et à copier les éléments du tableau à chaque fois qu'une insertion se produit mais plutôt lorsque la capacité est dépassée. Ceci explique le principe de la complexité amortie.
On aura alors le tableau suivant aprés l'insertion:

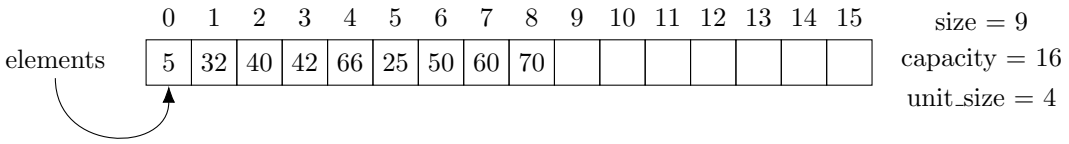

Figure 2 - Réallocation et recopie des éléments déja présents puis l'ajout de 50, 60 et 70

Les Questions :