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 le 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 $capcity * unit\_size$.

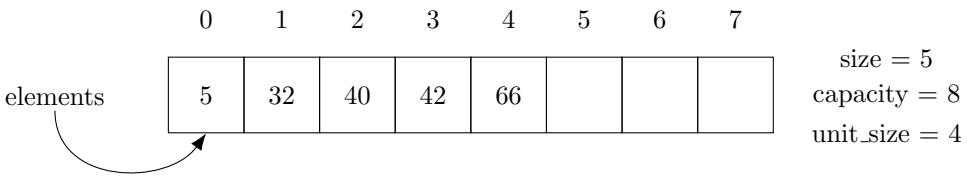

Figure 1 - Un exemple d'un `vector` de `int`

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

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

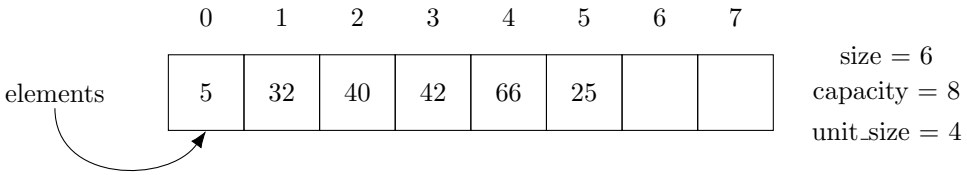

Figure 2 - Ajout de l'élément 25. Aucune allocation n'a eu lieu

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 :

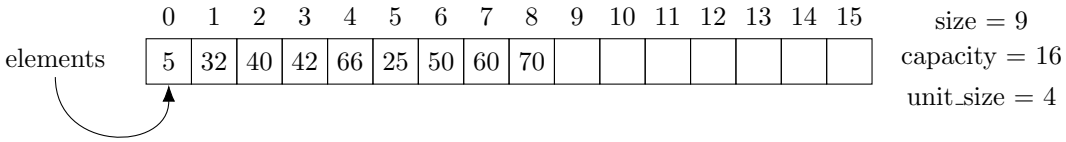

Figure 2 - Réallocation et recopie des éléments déja présents puis l'ajout de 50, 60 et 70

Les Questions :

Lisez l'intégralité du document avant d'écrire le code.

Question 1

Obtenez les fichiers `vector.h`, `main.c` et `Makefile` fournis dans le repistory GitHub.

Question 2

Vous devez implémentez les fonctions suivantes dans l'ordre, dans un fichier `vector.c`.

1- Impléméneter la fonction `init(vector* vec, size_t uint_size)` qui initalise un vecteur en mettant le champs capacité à `VECTOR_INIT_HALF_CAPACITY`. Cette fonction ne doit pas allouer de la mémoire.

```
/**
* Initilise the vector struct by setting :
* unit_size to the unit_size given in the arguments
* elements array to NULL (empty)
* size to 0
* capacity to VECTOR_INIT_HALF_CAPACITY
**/
void init(vector* vec, size_t unit_size);
```

2- Impléméneter la fonction `size(vector* vec)` qui retourne la taille du vecteur.

```
/**
* returns the size of vec
**/
size_t size(vector* vec);
```

3- Impléméneter la fonction `capacity(vector* vec)` qui retourne la capacité du vecteur.

```
/**
* returns the capacity of vec
**/
size_t capacity(vector* vec);
```

4- Impléméneter la fonction `reserve(vector* vec, size_t capacity)`. Si la capacité du vecteur est inférieure à la capacité donnée ou lorsque le champ `elements` n'est pas encore alloué (c'est-à-dire `NULL`) Cette fonction

fixe la capacité du vecteur à son double. Si la nouvelle capacité est encore plus petite que celle donnée en argument, alors elle affecte la capacité du vecteur pour que ca soit égale à celle donnée en argument. Ensuite, elle procède à l'allocation de mémoire. Sinon ça ne fait rien. Cette fonction servira ensuite comme une fonction d'utilité.

```
/**
* If vec->capacity is less then the given capacity or vec->elements is NULL then
*      vec->capacity is updated to be vec->capacity * 2
*      if capacity is still less than vec->capacity then vec->capacity must be
    capacity
*      vec->elements must now point to a newly allocated memory that can contain vec->
    capacity elements
*      the elemens existing in the older vec->elements must be copied over to the new
    vec->elements
* Otherwise nothing happens
* */
void reserve(vector* vec, size_t capacity);
```

5- Impléméneter la fonction `resize(vector* vec, size_t size))`. Cette fonction change la taille du vecteur pour qu'elle soit la même que la taille donnée dans les arguments. Si la capacité n'est pas suffisante, elle alloue la mémoire nécessaire et copie du l'ancien contenu dans la nouvelle mémoire. Vous pouvez l'implémenter facilement en utilisant la fonction réserver. La différence entre ces deux fonctions est que la réserve modifie la capacité tandis que cela modifie le nombre d'éléments présents dans le tableau (le champs `size`).

```
/**
* Sets vec size to the size given in the argument
* If the given size exceeds vec->capacity then
* elements must be reallocated to be able to contain
* (2 * size) elements. The elements already present
* must be copied to the new array.
* */
void resize(vector* vec, size_t size);
```

6- Impléméneter la fonction `get(vector* vec, size_t index)` qui retourne un pointeur vers l'élément qui se situe à la case avec l'indice `index`.

```
/**
* return a pointer the element at the index given by the argument
* if the index is out of bound an assertion occurs
* */
void* get(vector* vec, size_t index);
```

7- Impléméneter la fonction `front(vector* vec)` qui retourne un pointeur vers le premier élément de `vecteur`.

```
/**
* returns a pointer to the first element in vec
* if vec is empty then it asserts
* */
void* front(vector* vec);
```

8- Impléméneter la fonction `back(vector* vec)` qui retourne un pointeur vers le dernier élément de `vecteur`.

```
/**
* returns a pointer to last element in vec
* if vec is empty then it asserts
* */
void* back(vector* vec);
```

9- Impléméneter la fonction `push_back(vector* vec, void* element)` qui ajoute l'élément `element` à la derniere case du `vec`. Attention, le contenu de l'élément doit être entièrement copié, il ne faut pas stocker le pointeur ! Cette fonction doit gérer le cas ou la capacité est dépasser (Aide : utiliser la fonction `reserve`)

```
/**
* Add the element given in the arguments to the last cell of vec
* The content of the argument element must be copied (dont just store the pointer)
* Also handles the case where capacity is exceeded
* */
void push_back(vector* vec, void* element);
```

10- Impléméneter la fonction `push_front(vector* vec, void* element)` qui ajoute l'élément `element` à la première case du `vec`. Attention, le contenu de l'élément doit être entièrement copié, il ne faut pas stocker le pointeur ! Cette fonction doit gérer le cas ou la capacité est dépasser (Aide : utiliser la fonction `reserve`)

```
/**
* Add the element given in the argments to the first cell of vec
* The content of the argument element must be copied (dont just store the pointer)
* Also handles the case where capacity is exceeded
* The already existing elements are shifted to the right by one cell
* */
void push_front(vector* vec, void* element);
```

11- Impléméneter la fonction `insert(vector* vec, size_t index, void* element)` qui ajoute l'élément `element` à la `index` ième case du `vec`.

```c
/**
* Inserts the element given in the argments to the cell of index 'index'
* The content of the argument element must be copied (dont just store the pointer)
* Also handles the case where capacity is exceeded
* The already existing elements that are the right of 'index' are shifted to the right
* by one cell
* */
void insert(vector* vec, size_t index, void* element);
```

12- Impléméneter la fonction `shrink_to_fit(vector* vec)` qui change la capacité du vecteur tout en libérant la mémoire qui n'est pas utilisé. A la fin on doit avoir `size` égal à `capacity`.

```c
/**
* Changes the capacity of vec to be the same as its size
* And frees all the extra memory (the memory that is not used)
* */
void shrink_to_fit(vector* vec);
```

13- Impléméneter la fonction `destroy(vector* vec)` qui libère la mémoire allouée par vec (si aucune mémoire n'est allouée, alors ne fait rien), change sa taille en 0 et affecte `elements` à `NULL`.

```c
/**
* frees the memory allocated by vec (if no memory is allocated then it does nothing)
* sets its size to 0
* sets the pointer elements to NULL
* */
void destroy(vector* vec);
```

14- Compilez ce que vous avez fait en utilisant la commande `make` puis testez votre code en utilisant `make run`.

Quelques remarques

Vous pouvez utiliser la fonction `void* memcpy (void* dest, const void* src, size_t n);` défini dans `string.h` pour copier la mémoire de `src` vers la `dst`.
Exemple :

```c
#include <string.h>

int main()
{
    char str1[] = "Hello world";
    char str2[256];

    // Copies the content of str1 to str2
    memcpy(str2, str1, strlen(str1) * sizeof(char)));
    // this also works:
    // memcpy(str2, str1, sizeof(str1));

    int arr1[] = {1, 2, 3, 4, 5};
    int arr2[50];
    // Copies the content of arr1 to arr2
    memcpy(arr1, arr2, sizeof(arr1));
    // this also works:
    //  memcpy(arr2, arr1, 5 * sizeof(int));

    // this will copy only the first 3 elements:
    // memcpy(arr2, arr1, 3 * sizeof(int));

    // This will copy the first 3 elements of arr1 to arr2 starting from index 2
    memcpy(arr2 + 2, arr1, 3 * sizeof(int));
    // same as above :
    // memcpy(&arr2[2], arr1, 3 * sizeof(int));

    // This will copy the first 2 elements of arr1 starting from index 2 to arr2
        starting from index 5
    memcpy(arr2 + 5, arr1 + 2, 2 * sizeof(int));
    // same as above :
    // memcpy(&arr2[5], &arr1[2], 2 * sizeof(int));
}
```

Pour finir

Si vous avez réussi à tout mettre en œuvre et que les tests ont réussi, félicitations, vous avez implémenté quelque chose qui est utilisé dans le monde réel. Vous pouvez implémenter plus de fonctions qui suppriment un élément à un index donné, suppriment le dernier élément, etc.
