Cours de programmation séquentielle

Tableaux de taille dynamique

1 Buts

— Utilisation de tableaux.
— Allocation dynamique de mémoire.
— Utilisation de fonctions de fonctions.

2 Énoncé

En `C` il existe deux façon de créer des tableaux.

1. Les tableaux statiques, stockés sur la pile, qui se déclarent comme
```
typedef int type;
int size = 10;
type tab[size];
```

2. Les tableaux dynamiques, stockés sur le tas, qui se déclarent comme
```
typedef int type;
int size = 10;
type *tab = malloc(size * sizeof(type));
```

Notez comme dans les deux cas les tableaux sont de type `type` (qui ici n'est rien d'autre qu'un entier).

Une fois ces tableaux créés il n'est pas possible de rajouter ou d'enlever des éléments aux tableaux de façon simple. Dans ce travail pratique, vous allez implémenter une structure `vector`, qui est un tableau dynamique très performant tout en étant flexible existant dans la librairie standard de tous les langages modernes (mais pas en `C`). Nous pourrions créer une structure complètement générique à l'aide du type `void *`, mais cela demanderait de gros efforts qui ne seront pas forcément récompensés. Nous "émulerons" la généricité à l'aide d'un `typedef`.

2.1 La structure `vector`

Pour ce faire, il faut déclarer une structure `vector`

```
#define VECTOR_INIT_CAPACITY 4

typedef int type;
```

```c
typedef struct vector {
    type *content; // actual content of the vector
    int capacity; // capacity allocated
    int length; // actual length
} vector;
```

qui contiendra:

1. Un entier représentant la capacité du vecteur.
2. Un entier représentant la longueur du vecteur.
3. Un pointeur d'entier contenant les données du vecteur.

Puis il faudra implémenter les fonctions suivantes:

1. La fonction `vector_init(vector *v)` qui initialise un vecteur avec une capacité `VECTOR_INIT_CAPACITY` (voir ci-dessus), une longueur nulle, et alloue le pointeur d'entiers à une taille de `VECTOR_INIT_CAPACITY` de "`type`".
2. Une fonction `vector_length(vector *v, int *length)` qui stocke la longueur d'un vecteur dans `length`.
3. Une fonction `vector_push(vector *v, type element)` qui ajoute `element` au bout du vecteur. Si la longueur du vecteur dépasse sa capacité, il faudra réallouer le contenu du vecteur pour qu'il fasse deux fois sa capacité courante. Pour réallouer le tableau `content`, utiliser la fonction `realloc()` (voir `man 3 realloc` pour plus d'informations).
4. La fonction `vector_pop(vector *v, type *element)` qui retire le dernier élément du vecteur passé en argument et le stocke dans `element`. Si la longueur du vecteur devient plus petite que le quart sa capacité, on réallouera `content` qui aura une longueur de la moitié de la capacité courante.
5. La fonction `vector_set(vector *v, int index, type element)` qui assigne la `index`-ème valeur du vecteur à la valeur de `element`.
6. La fonction `vector_get(vector *v, int index, type *element)` qui copie le `index`-ème élément du vecteur dans `element`.
7. La fonction `vector_remove(vector *v, int index)` qui retire le `index`-ème élément du vecteur (attention à bien décaler tous les éléments du vecteur se trouvant après `index`).
8. La fonction `vector_insert(vector *v, type element, int index)` qui insère `element` au `index`-ème indice du vecteur (attention à bien décaler tous les éléments du vecteur se trouvant après `index`).
9. La fonction `vector_empty(vector *v)` qui vide un vecteur.
10. La fonction `vector_free(vector *v)` qui libère la mémoire du vecteur.

Puis implémenter également deux fonctions un peu plus complexes syntaxiquement.

11. La fonction `vector_map(vector *v, type (*f)(type), vector *rhs)` qui itère sur tous les élément du vecteur `v`, leur applique la fonction `f`, et stocke le résultat dans `rhs`.

12. La fonction `vector_filter(vector *v, bool (*f)(type), vector *rhs)` applique le prédicat `f` sur tous les éléments d'un vecteur et stocke ceux qui le satisfont dans le vecteur `rhs`.

Afin d'utiliser les fonctions `vector_map()` et `vector_filter()` vous devez écrire deux fonctions. La première, `type square(type elem)`, calculera le carré d'un élément. La seconde, `bool is_even(type elem)`, vérifiera si `elem` est pair. Vous pouvez implémenter d'autres fonctions si vous le souhaitez.

2.2 La gestion des erreurs

Chacune des fonctions ci-dessus peut échouer: une allocation mémoire pourrait échouer, on pourrait tenter d'enlever un élément inexistant, ... Afin de gérer ces erreurs au mieux, toutes les fonctions doivent retourner un code d'erreur:

```
typedef enum error_code {
    ok, out_of_bounds, memory_error, uninitialized
} error_code;
```

Afin de vous aider à débugger ou à vérifier les problèmes possibles implémentez une fonction qui lit le code d'erreur et retourne un message d'erreur.

2.3 Tests

Dans un premier temps écrivez vos propres tests pour vérifier que vos fonctions fonctionnent de la façon désirée. Dans un deuxième temps, nous vous fournirons une batterie de tests à nous également.

2.3.1 Tests automatiques

Comme lors du travail pratique précédent, nous vous fournissons gracieusement une batterie de tests pour ... tester vos implémentations. Pour ce faire vous devez, comme pour le TP sur les matrices, commencez par forker, configurer et cloner ce projet comme indiqué dans le document "Fork et clonage d'un dépôt GIT" sur cyberlearn. Le dépôt se trouve à l'adresse https://githepia.hesge.ch/programmation_sequentielle/travaux_pratiques/c_lang/c_vector_ci.

Dans ce dépôt vous trouverez:

1. les fichiers relatifs aux code d'erreur:
   — `error.h` qui contient le type énuméré avec les codes d'erreur.
   — `error.c` qui est vide et que vous pouvez remplir avec des fonctions affichant les erreurs.
2. les fichiers relatifs à la structure `vector`:
   — `vector.h` contenant les entêtes.
   — `vector.c` contenant des implémentations bidon (reprenez les implémentations que vous avez faites la dernière fois).
3. les fichiers relatifs aux fonctions pour le `vector_map` et `vector_filter`:
   — `map_functions.h` et `map_functions.c` contenant la fonction `square()`.
   — `filter_functions.h` et `filter_functions.c` contenant la fonction `is_even()`.

4. le fichier `main.c` qui contient une implémentation bidon de `main()` ne faisant rien.
5. le répertoire `tests` contenant tous les tests à passer. Vous ne devez pas modifier ces tests évidemment.