Cours de programmation séquentielle

Tableaux de taille dynamique

1 Buts

- Utilisation de tableaux.
- Allocation dynamique de mémoire.
- Utilisation de fonctions de fonctions.
- Émulation de généricité.

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 grâce au `typedef`).

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;
```

```
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 de type `type` contenant les données du vecteur. On suppose que `type` est un type de base de C (`double`, `float`, `int`, etc). Cela émule un comportement générique.

Important! La structure vector doit être un type **opaque**.

Puis il faudra implémenter les fonctions suivantes:

1. La fonction `vector_create()` qui crée un vecteur avec une capacité `VECTOR_INIT_CAPACITY` (voir ci-dessus), de longueur nulle, et alloue le pointeur d'entiers à une taille de `VECTOR_INIT_CAPACITY` de "`type`". S'il y a une erreur lors de l'allocation le programme doit planter.
2. Une fonction `vector_length(vector *v)` qui retourne la longueur d'un vecteur (pas sa capacité donc).
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). S'il y a une erreur d'allocation le programme doit planter.
4. La fonction `vector_pop(vector *v)` qui retire le dernier élément du vecteur et le retourne. 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. Si le vecteur est vide et qu'on essaie de `pop` le programme doit planter.
5. La fonction `vector_set(vector *v, int index, type element)` qui assigne la `index`-ème valeur du vecteur à la valeur de `element`. Si `index` n'est pas un indice valide le programme doit planter.
6. La fonction `vector_get(vector *v, int index)` qui retourne le `index`-ème élément du vecteur. Si `index` n'est pas un indice valide, le programme doit planter.
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`) et le retourne. Si `index` n'est pas un indice valide, le programme doit planter.
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`). Si `index` n'est pas un indice valide, le programme doit planter.
9. La fonction `vector_empty(vector *v)` qui vide un vecteur et le met dans le même état qu'après avoir été créé (voir `vector_create()`).

10. La fonction `vector_free(vector *v)` qui libère la mémoire du vecteur.
11. La fonction `vector_print(vector *v, void (*print)(type))` qui affiche le contenu du vecteur. Notez qu'il faut un pointeur de fonction pour faire l'affichage, comme le type de données à afficher peut changer. Cette fonction est importante pour faire du debugging.

Puis implémenter également trois fonctions un peu plus complexes syntaxiquement.

12. La fonction `vector_map(vector *v, type (*f)(type))` qui itère sur tous les élément du vecteur `v`, leur applique la fonction `f()`, et retourne un nouveau vecteur avec ces nouveaux éléments (ceux obtenus après application de la fonction `f()`).

```
int times_two(type a) {
    return 2 * a;
}
// retourne un nouveau vecteur où tous les éléments sont
// ceux de v, mais multipliés par 2.
vecteur *two_v = vector_map(v, times_two);
```

13. La fonction `vector_filter(vector *v, bool (*f)(type))` applique le prédicat `f` sur tous les éléments d'un vecteur et retourne un vecteur avec ces éléments.

```
bool lower_than_five(type a) {
    return (a < 5);
}
// retourne un nouveau vecteur contenant uniquement les éléments plus
// petits que 5
vector *v_lower_than_five = vector_filter(v, lower_than_five);
```

14. La fonction `vector_reduce(vector *v, type neutral, type (*f)(type, type))` applique la réduction `f` sur tous les éléments d'un vecteur et retourne l'élément réduit.

```
type add(type lhs, type rhs) {
    return lhs + rhs;
}
// retourne la somme de tous les éléments de v
// note: la somme commence à 0
type sum = vector_reduce(v, 0, add);
```

Dans tous les cas le vecteur `v` en argument n'est **jamais** modifié. Afin d'utiliser les fonctions `vector_map()`, `vector_filter()`, et `vector_reduce()`, vous devez écrire trois 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, la troisième `type mul(type lhs, type rhs)` calculera le produit de `lhs` et `rhs`. Vous pouvez implémenter d'autres fonctions si vous le souhaitez.

2.2 Remarque

Les points 11, 12, 13, et 14 nécessitent l'utilisation de **pointeurs de fonctions** que nous avons vus il y a plusieurs séances. Par exemple, en argument de la fonction `vector_reduce` nous avons en argument

```
type (*f)(type, type)
```

ce qui se lit comme une fonction prenant 2 arguments de type `type` et retournant un `type`. En fait le type de cet argument est

```
type (*)(type, type)
```

et le nom de la variable associée est `f`.

La fonction `add()` ci-dessus est un exemple d'un tel type

```
type add(type lhs, type rhs);
```

Si vous le souhaitez, vous pouvez utiliser un `typedef` pour clarifier. Vous aurez donc pour la signature de `vector_reduce()`

```
typedef type (*reduce_t)(type, type);
vector_reduce(vector *v, type neutral, reduce_t f)
```

2.3 La gestion des erreurs

La gestion des erreurs est presque la plus "simple" possible. Elle consiste à ne pas gérer les erreurs, mais à faire planter le programme. Pour atteindre ce but, le plus simple est d'utiliser des `assert()` (même si les puristes seront pas d'accord car ces erreurs peuvent être désactivée, mais dans le cadre de ce cours privilégions la simplicité).