P31 Développement efficace - épisode 2

1 - Un `vector` générique

Le `vector` du sujet précédent est écrit spécifiquement pour stocker un ensemble de valeurs entières.

L'objectif est maintenant d'écrire une structure de données et les fonctions associées pour un `vector` capable de stocker de manière contiguë un ensemble d'éléments de taille arbitraire.

En plus des informations que l'on maintenait dans le cadre du `vector` spécialisé pour des valeurs entières, ce `vector` générique devra mémoriser la taille en octets du type des éléments stockés. On appellera cette information `value_size`.

> ***Créer les fichiers `containers/vector.h` et `containers/vector.c`.***

Sur le même modèle que précédemment, le fichier `.h` contiendra la définition de la structure et les déclarations des fonctions et le fichier `.c` contiendra le corps de ces fonctions.

> ***Définir l'ensemble des fonctions suivantes :***

- initialisation et libération :

```
void vec_init(vector *v, size_t value_size);
void vec_free(vector *v);
```

- accès aux propriétés courantes :

```
size_t vec_size(vector const *v);
size_t vec_value_size(vector const *v);
size_t vec_capacity(vector const *v);
```

- réservation de mémoire :

```
void vec_reserve(vector *v, size_t nb_values);
```

- manipulation des données :

```
// retourne un pointeur sur la case d'index index
void const *vec_get_pointer(vector const *v, size_t index);
// copie la valeur contenue dans la case d'index index à l'adresse pointée par value
void vec_get_value(vector const *v, size_t index, void *value);
// copie la mémoire pointée par value dans la case d'index index
void vec_set_value(vector *v, size_t index, void const *value);
// copie la mémoire pointée par value dans une nouvelle case à la fin du vector v
void vec_push_back(vector *v, void const *value);
// copie la mémoire pointée par value dans une nouvelle case au début du vector v
void vec_push_front(vector *v, void const *value);
void vec_erase(vector *v, size_t index);
```

- échange de donnée, recherche, ...

```
void vec_swap(vector *v, size_t index1, size_t index2);
size_t vec_find(vector const *v, void const *value,
                bool (*equals)(void const *a, void const *b));
```

La fonction `vec_find` codée précédemment pour le type `vector_int` pouvait comparer directement les éléments avec l'opérateur de test d'égalité entre entiers `==`.

Avec un `vector` générique, on ne peut plus coder en dur la manière de tester l'égalité entre les éléments. On peut en revanche déléguer ce test à une fonction dont on reçoit un pointeur. Le paramètre supplémentaire

`equals` est ici un tel pointeur de fonction : appelée avec les adresses de deux éléments du vector, cette fonction retourne vrai si les éléments sont égaux et faux sinon.

Par exemple, si on a un `vector` de `float` déclaré et initialisé comme suit :

```
vector v;
vec_init(&v, sizeof(float));
float f = 1.5f;
vec_push_back(&v, &f);
f = 3.14f;
vec_push_back(&v, &f);
f = 0.42f;
vec_push_back(&v, &f);
```

Alors pour déterminer l'égalité entre deux éléments, on pourra fournir la fonction suivante :

```
bool equals_float(void const *a, void const *b)
{
    float f1 = *(float const *)a;
    float f2 = *(float const *)b;
    return f1 == f2;
}
```

Et appeler la fonction `vec_find` ainsi :

```
float p = 3.14f;
size_t i = vec_find(&v, &p, equals_float); // i vaut 1
```

> ***Dans les fichiers `utils.h` et `utils.c`, déclarer et définir des fonctions utilitaires comme `equals_int`, `equals_float`, etc..***

---

2 - Algorithme générique

Dans le même ordre d'idée, la précédente version de la fonction `bubble_sort_vec_int` pouvait comparer les éléments en utilisant directement l'opérateur de comparaison d'entiers `<`.

Avec un `vector` générique, on ne peut plus coder en dur la manière d'effectuer la comparaison entre les éléments.

> ***Dans le fichier `algos/bubble_sort.h`, ajouter la déclaration de la fonction suivante :***
>
> ```
> void bubble_sort_vec(vector *v, int (*compare)(void const *a, void const *b));
> ```

Le paramètre `compare` est ici un pointeur de fonction qui prend en paramètre les adresses de deux éléments du `vector` et retourne un entier négatif si le premier élément est inférieur au second, un entier positif si le premier élément est supérieur au second et 0 si les deux éléments sont égaux.

> ***Dans le fichiers `algos/bubble_sort.c`, écrire le corps de cette fonction.***

---

3 - Quelques tests

> ***Tester le bon fonctionnement du `vector` et de la fonction de tri avec plusieurs types primitifs tels que `int`, `float`, `double`.***

À chaque fois, il faudra écrire la fonction de comparaison correspondante (par exemple, `int compare_int(void const *a, void const *b)`) dans les fichiers `utils.h` et `utils.c`.

Afin de pouvoir tester le `vector` et la fonction de tri sur autre chose que sur des types primitifs, on va définir une nouvelle structure.

> ***Dans le fichier `utils.h`, définir la structure `rectangle` qui contient deux champs `width` et `height` de type `double`. Déclarer et écrire deux fonctions de comparaison différentes :***
> - ***`compare_rectangle_area` qui compare deux rectangles par leur aire***
> - ***`compare_rectangle_perimeter` qui compare deux rectangles par leur périmètre***

Une fois cela fait, dans votre `main`, déclarer un `vector` de `rectangle` et y stocker quelques rectangles. Trier ensuite ce `vector` avec les deux fonctions de comparaison en affichant le contenu du `vector` avant et après chaque tri.