Structures de données
TP4

Introduction

Dans les TPs précédants, vous avez développez en C une implémentation de la classe C++ « `Vector` » (documentation ici) permettant de stocker uniquement des `double`.

L'implémentation actuel n'est pas pratique car elle est dépendante du type de donnée stocker dans le `Vector`. En effet, si l'on veux changé le type de donnée manipulée, il est nécessaire de réécrire une grande partie des fonctions, ce qui n'est pas efficace.

Dans ce TP, vous allez développer en C une implémentation de la classe C++ « `Vector` » qui est indépendant du types de donnée, On parle d'**abstraction** au type.

Abstraction

Lors de l'implémentation d'une structure de données, il est intéressent quelle ne sois pas dédiée à un type de donnée spécifique (par exemple :`int`, `float`, `char *`, `struct my_struct`, ...), l'implémentation d'une structure de données doit faire **abstraction** du type de donnée quelle manipule.

Dans une structure de donnée, comme celle que vous implémentez, la structure n'effectue que 3 type d'opération du les données :

— l'allocation ;
— la suppression ;
— la copie.

Pour créer une structure de donnée universelle, indépendante du type de donnée manipulé, il suffi de crée une structure de données avec un type générique et de lui fournir les 3 fonctions pour manipuler un type de donnée spécifique lors de son utilisation.

En C, la solution la plus simple pour implémenter des structures de données qui font **abstraction** du type de donnée manipulé est d'utilisation du type `void *`. En effet, `void *` est un pointeur dont le type de donnée pointé est non-définie. Vous pouvez alors développer des structures de données qui ne stocke pas directement la donnée mais un pointeur non-typé pointant sur la donnée.

1 Structure `my_struct`

Dans un premier temps, vous allez créer une petite structure de donnée que vous utiliserez plus tard pour tester votre implémentation générique de la classe C++ « `Vector` ». Cette structure doit contenir :

— une chaîne de caractères alloué dynamiquement ;
— un nombre de type `double`.

1.1. Créez dans votre répertoire de travail les fichiers

— « `my_struct.h` » le fichier d'entête de votre structure de test ;
— « `my_struct.c` » le fichier de code de votre structure de test ;
— « `test_my_struct.c` » le fichier de testes unitaires de votre structure de test (contient le `main`).

(a) Ajoutez dans le fichier « `my_struct.h` », les instruction de précompilation pour sécuriser votre fichier contre les doubles inclusions.

(b) Ajoutez dans les fichiers « `my_struct.c` » et « `test_my_struct.c` », l'inclusion du fichier « `my_struct.h` ».

(c) Ajoutez dans le fichier « `test_my_struct.c` », la fonction `main`.

(d) Ajoutez dans le fichier « `makefile` », les lignes pour compiler le fichier « `my_struct.c` » et obtenir le fichier « `my_struct.o` ».

(e) Ajoutez dans le fichier « `makefile` », les lignes pour compiler le fichier « `test_my_struct.c` » et obtenir le fichier « `test_my_struct.o` ».

(f) Ajoutez dans le fichier « `makefile` », les lignes pour compiler le fichier « `test_my_struct.c` » et obtenir le fichier « `test_my_struct.o` ».

(g) Ajoutez dans le fichier « `makefile` », les lignes pour compiler l'exécutable final « `test_my_struct` » a partir des fichiers « `test_my_struct.o` » et « `my_struct.o` ».

(h) Testez en exécutant la commande `make test_my_struct` dans votre répertoire de travail.

1.2. Définition de la structure

(a) Déclarez dans le fichier « `my_struct.h` » votre structure, que vous nommerez « `struct_my_struct` ».

(b) Utilisez la commende C `typedef` (doc. ici) pour redéfinir votre `struct struct_my_struct` en `s_my_struct` et pour définir le type `p_s_my_struct` qui est un pointeur sur la structure de `s_my_struct`.

1.3. Fonctions pour votre structure.

(a) Écrivez la fonction `p_s_my_struct my_struct_alloc();` qui alloue votre structure vide et la retourne.

(b) Écrivez la fonction `void my_struct_free(p_s_my_struct p_vector);` qui libère votre structure.

(c) Écrivez la fonction `void my_struct_randoms_init(p_s_my_struct p_vector);` qui initialise de manière aléatoire le contenu de votre structure.

(d) Écrivez la fonction `void my_struct_copy(p_s_my_struct p_dest, p_s_my_struct p_src);` qui recopie le contenue de la structure `p_src` dans la structure `p_dest`.

(e) Écrivez la fonction `int my_struct_cmp(p_s_my_struct p_vector_a, p_s_my_struct p_vector_b );` qui compare les structures et qui retourne :
— **0** si `p_vector_a` est identique à `p_vector_b` ;
— **1** si `p_vector_a` dois être classé après `p_vector_b` ;
— **-1** si `p_vector_a` dois être classé avant `p_vector_b`.

1.4. Fichier de test

(a) Créez le fichier « `test_my_struct.c` ».

(b) Ajoutez dans ce fichier le code minimum pour la fonction `int main(int argc, char *argv[])`.

(c) Ajoutez dans le fichier « `makefile` », les lignes pour compiler le fichier « `test_my_struct.c` » et obtenir le fichier « `test_my_struct.o` »

(d) Testez en exécutant la commande `make test_my_struct.o` dans votre répertoire de travail.

(e) Ajoutez dans le fichier « `makefile` », les lignes pour effectuer le linkage de « `test_my_struct.o` » et « `my_struct.o` » et obtenir le fichier « `test_my_struct` »

(f) Testez en exécutant la commande `make test_my_struct` dans votre répertoire de travail.

(g) Dans le fichier « `test_my_struct.c` » écrivez des fonctions de teste unitaire pour tester toute vos fonctions de votre vecteur.

2 Structure `vector`

Maintenant, vous allez modifier votre implémentation de la classe C++ pour la rendre indépendante du type de donnée manipulé.

La première étape va consisté à modifié le type de donnée que votre structure de données `vector` gère. Votre structure actuel manipule des `double`, vous allez la modifier pour quelle manipule des `void *`

2.1. Créez une copie de tous les fichiers de votre structure de données vector_v2_double (`vector_v2_double.c`, `vector_v2_double.h`, `test_vector_v2_double.c`, ...) et renommez les en **vector**, renommez également toutes les structures et fonctions « `vector_v2_double` » en « `vector` ».

2.2. Ajoutez dans le fichier « `makefile` », les lignes pour compiler l'ensemble des fichiers de `vector`.

2.3. La première étape va consisté à modifié le type de donnée que votre structure de données `vector` gère. Votre structure actuel manipule des `double`, vous allez la modifier pour quelle manipule des `void *`

(a) Modifiez la structure pour gérer des `void *`, vous aurez donc un pointeur de `void **` pour avoir un tableau de `void *`.

(b) Modifiez la structure pour gérer des `void *`, vous aurez donc un pointeur de `void **` pour avoir un tableau de `void *`.

(c) Modifiez en fonction la fonction `p_s_vector vector_alloc(size_t n);`. Tout les pointeurs doivent être initialisés à `NULL`.

(d) Modifiez les fonctions d'accès (`vector_set`, `vector_get`, `vector_insert`, `vector_erase`, `vector_push_back` et `vector_pop_back`) pour quelles prennent en paramètres ou en retours des `void *`.

(e) Modifiez votre programme de test (`test_vector_v2_double.c`), vous pouvez utiliser la fonction la fonction `size_t random_size_t(size_t a, size_t b);` pour générer des fausses adresses.

2.4. Avec les modifications que vous venez d'apporter, votre structure de données `vector` vous permet de stocker des pointeurs sur n'importe quelle type de donnée. Cela est un début mais ce n'est pas encore satisfaisant. En effet, avec cette structure de donnée, vous pouvez gérer n'importe quelle type de donnée mais comme elle n'est capable que de gérer des pointeurs, c'est à l'utilisateur de gérer la mémoire (l'utilisateur dois faire l'allocation des données pour ensuite fournir les pointeurs sur les données à la structure.).

Pour remédier à cela, vous devez donner à votre structure `vector` les fonctions pour quelle gère elle même la mémoire :

— `void * data_alloc();` qui permet d'allouer une donnée ;

— `void data_free(void * p_data);` qui permet de libérer une donnée ;

— `void data_cpy(void *p_data_dst, void *p_data_src);` qui permet de faire une copie de la donnée `p_data_src` vers `p_data_dst`.

(a) Ajoutez dans le fichier « `vector.h` » la définition des 3 types de pointeurs de fonctions suivants :
   - `typedef void * (* t_data_alloc)();`
   - `typedef void   (* t_data_free)(void * p_data);`
   - `typedef void   (* t_data_cpy)(void *p_data_dst, void *p_data_src);`

(b) Modifiez la structure `struct_vector` pour y ajouter les trois pointeurs de fonctions que vous avez défini précédemment.

(c) Modifiez la fonction `p_s_vector vector_alloc(size_t n);` pour y passer en argument les trois pointeurs de fonctions (allocation, libération et copie de donnée). Et stocker ces pointeurs dans la structure `struct_vector`. Utilisez le pointeur sur la fonction d'allocation pour allouer les éléments du vecteur au lieu d'initialisé le tableau de pointeur avec la valeur `NULL`.

(d) Modifiez les fonctions `vector_free` et `vector_clear`, utilisez la fonction de libération des données pour libérer les donnes que vous n'utiliserez plus.

(e) Modifiez la fonction `vector_set`, lors de l'affectation d'une valeur dans le vecteur, vous ne devez plus copier l'adresse passé en paramètre dans le tableau de `void *` mais vous devez utiliser la fonction de copie de donnée pour recopier la donnée passé en paramètre dans la $i^{\text{ème}}$ de votre tableau.

(f) Pour la fonction `vector_get`, vous allez modifier le prototype de la fonction. En effet, comme vos structure vector stocke des copies des données, l'utilisateur ne dois pas avoir directement accès aux éléments stockés, vous n'allez donc plus retourner directement le pointeur sur un des éléments stocké mais une copie de cette élément. Pour cela, nous allons faire un retourne par les arguments de la fonction. Le nouveau prototype est alors : `void vector_get(p_s_vector p_vector, size_t i, void * p_data);`. Modifiez alors la fonction pour quelle recopie la $i^{\text{ème}}$ données du vecteur dans le pointeur `p_data`, passé en paramètre de la fonction.

(g) Avec les mêmes principes, modifiez les fonctions `void vector_insert(p_s_vector p_vector, size_t i, double v);` et `void vector_erase(p_s_vector p_vector, size_t i);`.

2.5. Modifiez vos fichiers « `test_vector.c` » et « `bench_vector.c` » pour tester votre nouvelle structure universelle, vous utiliserez la structure `my_struct` que vous avez développé dans la première partie de ce TP pour testé votre structure `vector`.