Structures de données
TP2

Introduction

Dans la suite des TPs de structure de données, vous allez développer une bibliothèque en C dans laquelle vous implémenterez :

- des structures de données :
  - Tableau dynamique
  - Liste doublement chaînée
  - Pile et File
  - ...
- des fonctions pour mesurer les performances de vos structures de données :
  - Mesure du temps d'exécution
  - Mesure du nombre d'allocations mémoire
  - Mesure de la quantité de mémoire allouée

Vous allez également développer des codes de test basés sur plusieurs algorithmes de tri pour évaluer les performances de vos implémentations.

Afin de garantir un code propre, robuste et facile à maintenir, plusieurs bonnes pratiques devront être suivies :

- **Tests unitaires** : Chaque fonction devra être testée de manière isolée pour garantir son bon fonctionnement. Des tests unitaires systématiques permettront de détecter rapidement les erreurs et éviter les régressions.
- **Compilation avec Makefile** : Vous devez écrire un `Makefile` permettant la compilation automatique de votre projet. Ce fichier devra inclure des règles pour compiler, nettoyer les fichiers intermédiaires et exécuter les tests.
- **Gestion des erreurs** : Il est crucial d'implémenter des mécanismes de gestion des erreurs pour éviter les comportements imprévisibles du programme. L'utilisation d'énumérations de codes d'erreur permettra d'améliorer la robustesse du projet.
- **Commentaires et documentation** : Un code bien commenté est plus lisible et compréhensible. Il est impératif d'expliquer le rôle des variables, des structures et des fonctions afin de faciliter leur compréhension par d'autres développeurs.

1 Découverte de l'environnement de travail

Avant de commencer l'implémentation des structures de données, il est important de se familiariser avec l'environnement de développement, la compilation et les outils qui seront utilisés tout au long du TP.

1.1 Compilation et Makefile

Le **Makefile** est un fichier qui automatise la compilation du projet. Il permet d'éviter d'avoir à exécuter manuellement les commandes de compilation en définissant des règles claires et en facilitant le débogage.

Utilisation de Makefile Pour compiler le projet, il suffit d'exécuter la commande :

```
make
```

Cela génère l'exécutable `test_unit` en compilant les fichiers sources automatiquement.

Exécution avec Valgrind Pour détecter les erreurs mémoire et les fuites de mémoire, utilisez :

```
make valgrind
```

Cela exécute le programme avec `valgrind` et affiche les éventuelles fuites mémoire.

Débogage avec GDB Pour exécuter le programme en mode débogage avec `gdb`, utilisez :

```
make debug
```

Cela permet d'examiner l'exécution du programme et de détecter des erreurs.

Nettoyage des fichiers compilés Pour supprimer les fichiers objets et l'exécutable, utilisez :

```
make clean
```

Pour un nettoyage complet incluant les fichiers temporaires, utilisez :

```
make distclean
```

Ce Makefile offre une gestion efficace de la compilation et du débogage, simplifiant ainsi le travail !

1.2 Gestion des erreurs

La gestion des erreurs est essentielle pour éviter que votre programme ne plante ou se comporte de manière imprévisible. Pour cela, nous avons introduit un système de codes de retour avec l'énumération suivante :

```
typedef enum {
    VECTOR_SUCCESS = 0,       // Opération réussie
    VECTOR_ERROR_NULL_PTR = -1,  // Pointeur NULL
    VECTOR_ERROR_OUT_OF_BOUNDS = -2, // Index hors limites
    VECTOR_ERROR_ALLOCATION = -3,  // Erreur d'allocation mémoire
    VECTOR_ERROR_EMPTY = -4    // Vecteur vide
} VectorStatus;
```

Ces codes permettent d'identifier rapidement la cause d'un problème et d'adopter une stratégie adaptée pour y remédier.

1.3 Tests unitaires

Les tests unitaires sont essentiels pour vérifier que vos fonctions se comportent correctement. Ils permettent de détecter des erreurs rapidement et facilitent la maintenance du code.

Un **test unitaire** est une fonction qui vérifie qu'une autre fonction retourne bien la valeur attendue pour un ensemble donné d'entrées.

Vous devez utiliser la macro suivante pour écrire vos tests :

```
#define ASSERT(condition) do { \
    if (!(condition)) { \
        fprintf(stderr, "Assertion échouée dans %s:%d: %s\n", __FILE__,
            __LINE__, #condition); \
        return 1; \
    } \
} while (0)
```

Cette macro permet d'afficher un message d'erreur et d'arrêter l'exécution du test en cas d'échec.

Exemple de test unitaire

```c
int test_vector_alloc() {
    p_s_vector v = vector_alloc(5);
    ASSERT(v != NULL);
    ASSERT(v->size == 5);
    vector_free(&v);
    ASSERT(v == NULL);
    return 0;
}
```

Pour exécuter vos tests unitaires, utilisez la commande :

```
./test_unit
```

2 Tableau dynamique

Dans ce TP, vous allez développer un tableau dynamique. C'est une structure de données qui permet :

- de stocker un ensemble de n données ;
- d'accéder aux données par un index ;
- d'ajouter et de supprimer dynamiquement des données.

Classe C++ : `Vector`

Pour développer votre structure de tableau dynamique, vous allez vous inspirer de la classe C++ « `Vector` » (documentation ici). Les principales fonctions de la classe Vector que nous voulons reproduire sont :

- `vector::at`
- `vector::erase`
- `vector::insert`
- `vector::push_back`
- `vector::pop_back`
- `vector::clear`
- `vector::empty`
- `vector::size`
- `vector::capacity`

Pour optimiser l'utilisation mémoire et éviter trop d'allocations dynamiques, vous implémenterez un mécanisme de gestion de capacité inspiré de la classe `Vector` en C++.

3 Travaux préalables

3.1. Créez votre répertoire de travail, nommez-le « `i3_in9_lib` ».

3.2. Recopiez dans votre répertoire de travail les fichiers :

- `Makefile` : contient les instructions pour la compilation automatique du projet ;
- `test_unit.c` : contient la fonction *main* et les tests unitaires ;
- `vector.c` : contient les fonctions de la structure du tableau dynamique ;
- `vector.h` : contient la définition de la structure et les prototypes des fonctions ;
- `README`.

3.3. Ouvrez un terminal dans votre répertoire de travail et exécutez les commandes suivantes :

(a) `make` : Compile automatiquement le projet. Vous ne devriez avoir que des `warnings` liés aux paramètres non utilisés ou aux `return` manquants.

(b) `./test_unit` : Exécutez le fichier de test.

(c) `make clean` : Nettoie tous les fichiers générés lors de la compilation.

Bonnes pratiques à respecter

- **Ajoutez des commentaires dans vos codes.**
- **Vérifiez les arguments passés aux fonctions** : pointeurs non `NULL`, indices valides, etc.
- **Utilisez des tests unitaires avec `ASSERT()`** pour vérifier le bon fonctionnement des fonctions.

4 Implémentation - Tableau dynamique de double

Dans un premier temps, vous allez développer une structure de tableau **dynamique simple** pour stocker des `double`. Cette première implémentation naïve consiste à allouer un tableau de taille n, et à réallouer dynamiquement la mémoire à chaque ajout ou suppression d'un élément.

Bien que fonctionnelle, cette approche est inefficace en raison des nombreuses réallocations de mémoire qu'elle engendre. Nous verrons dans les TPs suivants comment optimiser la gestion de la mémoire afin d'éviter ces réallocations coûteuses.

Les types `size_t` et `ssize_t`

Le type `size_t` en C est un entier non signé utilisé pour représenter la taille d'objets et de tableaux. Il assure une compatibilité entre différentes architectures (32 bits ou 64 bits) et est couramment utilisé dans des fonctions comme `malloc`, `strlen` et `sizeof`.
Cependant, `size_t` ne peut pas représenter de valeurs négatives, ce qui peut être problématique lorsqu'on manipule des indices ou lorsqu'une fonction doit signaler une erreur. C'est pourquoi `ssize_t`, une version signée de `size_t`, est souvent utilisée.
Dans notre implémentation du `vector`, nous utilisons :

- `size_t` pour représenter la taille totale du tableau dynamique, car elle ne peut jamais être négative.
- `ssize_t` pour les indices des tableaux et les retours de fonction, permettant ainsi d'indiquer des erreurs avec des valeurs négatives (exemple : `-1` pour un indice invalide).

L'utilisation de `ssize_t` permet une meilleure gestion des erreurs et rend notre code plus robuste.

4.1. Structure pour votre tableau dynamique.

(a) Dans le fichier « `vector.h` », complétez la structure nommée « `struct_vector` ». Elle doit contenir :
- Une variable pour stocker le nombre d'éléments stockés dans la structure.
- Un pointeur de `double` pour contenir les données.

(b) Ajoutez des commentaires sur les deux lignes de `typedef` (doc. ici) situées au-dessous de la déclaration de la structure pour expliquer leur utilité.

(c) Complétez la fonction `p_s_vector vector_alloc(size_t n);` qui alloue et initialise un tableau dynamique contenant `n` valeurs `double` initialisées à 0.0.

(d) Complétez la fonction `VectorStatus vector_free(p_s_vector *p_vector);` qui libère la mémoire et met le pointeur de la structure à `NULL`.

(e) Complétez la fonction `VectorStatus vector_set(p_s_vector p_vector, ssize_t i, double v);` qui affecte une valeur à un index donné, et `VectorStatus vector_get(p_s_vector p_vector, ssize_t i, double *pv);` qui récupère la valeur stockée à un index donné.

(f) Complétez la fonction `VectorStatus vector_insert(p_s_vector p_vector, ssize_t i, double v);` qui insère une nouvelle donnée à l'index `i` de votre tableau dynamique.
Cette fonction est l'équivalent de la méthode `vector::insert` de la classe C++ `Vector`.

(g) Complétez la fonction `VectorStatus vector_erase(p_s_vector p_vector, ssize_t i);` qui supprime la donnée à l'index `i` de votre tableau dynamique.
Cette fonction est l'équivalent de la méthode `vector::erase` de la classe C++ `Vector`.

(h) Complétez la fonction `VectorStatus vector_push_back(p_s_vector p_vector, double v);` qui insère une nouvelle donnée à la fin de votre tableau dynamique.
Cette fonction est l'équivalent de la méthode `vector::push_back` de la classe C++ `Vector`.

(i) Complétez la fonction `VectorStatus vector_pop_back(p_s_vector p_vector);` qui supprime la dernière donnée de votre tableau dynamique.
Cette fonction est l'équivalent de la méthode `vector::pop_back` de la classe C++ `Vector`.

(j) Complétez la fonction `VectorStatus vector_clear(p_s_vector p_vector);` qui supprime toutes les données de votre tableau dynamique.
Cette fonction est l'équivalent de la méthode `vector::clear` de la classe C++ `Vector`.

(k) Complétez la fonction `VectorStatus vector_empty(p_s_vector p_vector);` qui retourne un `VectorStatus`.
Cette fonction est l'équivalent de la méthode `vector::empty` de la classe C++ `Vector`.

(l) Complétez la fonction `VectorStatus vector_size(p_s_vector p_vector, size_t *size);` qui stocke le nombre d'éléments dans `size` et retourne un `VectorStatus`.
Cette fonction est l'équivalent de la méthode `vector::size` de la classe C++ `Vector`.