Structures de données
TP4

Introduction

Dans le TP précédant, vous avez développez en C un implémentation simple de la classe C++ « `Vector` » (documentation ici) permettant de stocker uniquement des `double`.

Dans ce TP, vous allez réorganisé vôtres répertoire de travail pour ne plus avoirs tout les fichiers au même endroits et vous allez modifier votre implémentassions pour avoir une gestions plus intelligente de la mémoire.

1 Réorganisation du répertoire

Dans l'archive du sujet du TP4, vous trouverez un nouveau répertoire de travail. Il est organisé de la magnerai suivante :

- `./src` : Répertoire contenant les fichiers source C du projet.
- `./headers` : Répertoire contenant les fichiers en-tête (header) du projet.
- `./objs` : Répertoire où sont générés les fichiers objet du projet.
- `./bin` : Répertoire où sont générés les exécutables du projet.
- `./coverage` : Répertoire où sont générés les rapports de couverture de code lorsque l'option COVERAGE est activée.

Vous y trouverez également un nouveau fichier « `makefile` ».

Documentation « `makefile` »

Options de compilation

Voici les options de compilation disponibles avec leur syntaxe et leur signification :

- `DEBUG=1` : Active le mode débogage.
- `COVERAGE=1` : Active la couverture de code. Seul l'exécutable meta_test est compilé.
- `VERSION=2` : Active la version 2.0 du code.
- `VERSION=3` : Active la version 3.0 du code.

Par exemple, pour activer le mode débogage et la version 2.0 du code, vous pouvez utiliser la commande suivante : `make DEBUG=1 VERSION=2`

Exécution des cibles

Voici les cibles disponibles avec leur syntaxe et leur signification :

- `all` : Génère tous les exécutables. C'est la cible par défaut lorsque vous exécutez la commande 'make' sans spécifier de cible.
- `clean` : Supprime les fichiers objet et exécutables générés.
- `info` : Affiche la valeur de certaines variables utilisées dans le makefile.

Par exemple, pour nettoyer le projet et afficher la valeur de certaines variables, vous pouvez utiliser les commandes suivantes : `make clean` et `make info`

Couverture de code

Lorsque l'option `COVERAGE` est activée, seul l'exécutable `meta_test` est compilé et exécuté. Ensuite, le rapport de couverture de code est généré et enregistré dans un répertoire `html_cov`, `html_v2_cov` ou `html_v3_cov` selon la version du code spécifiée avec l'option `VERSION`. Voici un exemple de commande pour activer la couverture de code avec la version 2.0 du code : `make COVERAGE=1 VERSION=2`

1.1. Recopiez dans le nouveau répertoire de travail « `i3_in9_lib` » vos fichiers de votre ancien répertoire de travail en les rangent dans le répertoire approprié.

1.2. Ouvrez un terminal dans votre répertoire de travail.

(a) Dans le terminal, tapez la commande `make`. Cette commande permet de compiler automatiquement le projet. La compilation génère les fichiers exécutables suivant : « `test_vector` », « `test_random` », « `bench_vector` » et « `conditional_compilation_demo` » dans le répertoire `bin`. Vous pouvez les exécutées par exemple avec la commande suivant : `./bin/test_vector`

(b) Tapez la commande `make DEBUG=1`. Cette commande permet de compiler automatiquement le projet en mode débogage (très utilise pour l'utilisation d'outils tel que `valgrind`). La compilation génère les fichiers exécutables suivant : « `test_vector_debug` », « `test_random_debug` », « `bench_vector_debug` » et « `conditional_compilation_demo_debug` » dans le répertoire `bin`. Vous pouvez les exécutées par exemple avec la commande suivant : `./bin/test_vector_debug`

(c) Tapez la commande `make VERSION=2` Cette commande permet de compiler automatiquement le projet avec la déclaration du FLAG VERSION=2 . La compilation génère les fichiers exécutables suivant : « `test_vector_v2` », « `test_random_v2` », « `bench_vector_v2` » et « `conditional_compilation_demo_v2` » dans le répertoire `bin`.

(d) Exécutez les exécutables :
- `./bin/conditional_compilation_demo` : Correspondant au fichier C « `conditional_compilation_demo.c` » compilé sans la déclaration du FLAG `VERSION`
- `./bin/conditional_compilation_demo_v2` : Correspondant au fichier C « `conditional_compilation_demo.c` » compilé avec la déclaration du FLAG `VERSION=2`

Le comportement de l'exécutable a changé, allez voir le code du fichier C « `conditional_compilation_demo.c` » pour comprendre pourquoi (Nous utiliserons ce mécanisme dans la suite du TP).

(e) Tapez la commande `make COVERAGE=1`. Cette commande compile le fichier C « `meta_test.c` » et crée un rapport de couverture de code de l'exécutable `meta_test_cov`. Vous pouvez consulté ce rapport en ouvrent le fichier `./coverage/html_cov/index.html`.

Couverture de code

La couverture de code en C est un outil très utile pour évaluer la qualité et la robustesse d'un logiciel. Elle permet de mesurer à quel point les différentes parties du code ont été testées lors de la réalisation des tests unitaires. Plus la couverture de code est élevée, plus le logiciel est considéré comme fiable et robuste. En utilisant des outils de couverture de code, il est possible de détecter les parties du code qui ne sont pas suffisamment testées et de les cibler lors de la réalisation de nouveaux tests. Ainsi, la couverture de code est un élément essentiel pour améliorer la qualité et la fiabilité d'un logiciel en C.

(f) Pour nettoyer votre répertoire de tous les fichiers générés lors de la compilation vous pouvez exécuter la commande : `make clean`

1.3. Vous devez, tout au long des TPs, **ajouter des commentaires dans vos codes**, vous devez également ajouté des commentaire sur les codes déjà fournie : sur la définition de la structure, sur les prototypes des fonctions, *etc.*

1.4. Vous devez, tout au long des TPs, **ajouter des tests sur les arguments passés aux fonctions** : vérifier qu'un pointeur n'est pas `NULL`, vérifier qu'un indice de tableau est bien dans la plage défini du tableau, *etc.*

2 Tableau dynamique V2.0

Maintenant, vous allez ajouter à votre structure de tableau dynamique, une stratégie pour éviter de ré-allouer le tableau à chaque insertion ou suppression d'un élément. Pour cela, nous allons utilisé deux variable :

— **size** : le nombre d'éléments « référencés » dans la structure ;trouverez un exemple de compilation conditionnelle dans le répertoire :
« `exemple_compilation_conditionnelle` »).

— **capacity** : le nombre d'éléments « référençables » dans la structure (taille réel du tableau alloué).

Les règles que nous allons suivante sont les suivantes :

1. La capacité minimum de notre structure est de 16 éléments (capacity ≥ 16.) ;
2. Quand l'utilisateur demande la création d'un vecteur de taille n, capacity initialisé a la puissance de 2 directement supérieur à n.
3. Après l'ajout d'un élément, si capacity = size alors vous doublez la capacité de votre structure (capacity $\leftarrow 2 \times$ capacity) ;
4. Après la suppression d'un élément, si size $\leq \frac{\text{capacity}}{4}$ alors vous divisez par 2 la capacité de votre structure (capacity $\leftarrow \frac{\text{capacity}}{2}$)

Pour implémenter cette nouvelle stratégie d'allocation, nous allons utiliser la compilation conditionnelle de code avec les instructions pré-processeur : `#if, #endif, ...`. Cela perpétra de faire co-habiter les deux stratégie dans le même fichier sans avoir à créer des fonctions spécifiques pour chaque stratégie et à simplement rajouter la déclaration d'un « FLAG » lors de la compilation pour choisir quelle version du code compiler. Le FLAG a déjà été définie dans le makefile (`make VERSION=2` voir la documentation du makefile plus haut). Vous trouverais un exemple de compilation conditionnel dans le fichier C : « `conditional_compilation_demo.c` ».

2.1. Dans `vector.h` en utilisant la compilation conditionnelle, modifiez la structure de votre tableau dynamique en y ajoutant une variable pour stocker la capacité.

2.2. Dans `vector.c` en utilisant la compilation conditionnelle, modifiez les fonctions de votre tableau dynamique pour prendre en compte les règles pour limité le nombre de réallocation.

2.3. Re-testez et regardé les performances après vos modification.