Structures de données
TP3

Introduction

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

Dans ce TP, vous allez développer des tests pour évaluer les performances, en temps et en mémoire, de votre implémentation.

1 Évaluation des performances

Pour évaluer les performances d'un tableau dynamique, vous mesurerez :
- le temps d'exécution ;
- le nombre d'allocations mémoire ;
- la quantité moyenne de mémoire allouée.

Dans les scénarios suivants :
- l'ajout et la suppression d'éléments à des index aléatoires ;
- l'ajout et la suppression d'éléments en tête et en queue ;
- l'accès aléatoire en lecture et écriture ;
- l'accès séquentiel en lecture et écriture ;
- tri des données.

2 Réorganisation du répertoire et Travaux préalables

Dans l'archive du sujet du TP4, vous trouverez un nouveau répertoire de travail. Il est organisé de la manière 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` ».

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

2.2. Vous trouverez également les fichiers suivants :
- « `./src/random.c` » : contient les fonctions de génération aléatoire ;
- « `./headers/random.h` » : contient la définition des prototypes des fonctions de génération aléatoire ;
- « `./src/bench_vector.c` » : contient la fonction *main* pour les tests de performances de la structure du tableau dynamique.

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 test_unit 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 `test_unit` 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`

2.3. 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 suivants : « `test_unit` », « `bench_vector` » et « `conditional_compilation_demo` » dans le répertoire `bin`. Vous pouvez les exécuter par exemple avec la commande suivante : `./bin/test_unit`

(b) Tapez la commande `make DEBUG=1`. Cette commande permet de compiler automatiquement le projet en mode débogage (très utile pour l'utilisation d'outils tels que `valgrind`). La compilation génère les fichiers exécutables suivants : « `test_unit_debug` », « `bench_vector_debug` » et « `conditional_compilation_demo_debug` » dans le répertoire `bin`. Vous pouvez les exécuter par exemple avec la commande suivante : `./bin/test_unit_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 suivants : « `test_unit_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_compil` compilé sans la déclaration du FLAG `VERSION`
- `./bin/conditional_compilation_demo_v2` : Correspondant au fichier C « `conditional_com` compilé avec la déclaration du FLAG `VERSION=2`

Le comportement de l'exécutable a changé, allez voir le code du fichier C « `conditional_compilat` 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 « `test_unit.c` » et crée un rapport de couverture de code de l'exécutable `test_unit_cov`. Vous pouvez

consulter ce rapport en ouvrant 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`

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

2.5. 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 définie du tableau, *etc.*

3 Générateur aléatoire

Dans un premier temps, vous allez développer un ensemble de fonctions pour générer des nombres et chaînes aléatoires.

En informatique, la génération de nombres aléatoires repose souvent sur des algorithmes de génération de nombres pseudo-aléatoires. En C, la fonction standard pour générer des nombres aléatoires est `int rand(void);`, définie dans `#include <stdlib.h>`. Cette fonction génère des nombres pseudo-aléatoires ; à chaque appel, elle retourne un nombre entier compris entre 0 et `RAND_MAX` (inclus).

Fonctionnement de `rand()`

La fonction `rand()` utilise un algorithme de génération de nombres pseudo-aléatoires, qui produit une séquence de nombres qui semblent aléatoires mais sont en réalité déterministes. La plage de valeurs possibles s'étend de 0 à `RAND_MAX`, une constante définie dans `stdlib.h` et dont la valeur est au moins 32767.

Initialisation du générateur avec `srand()`

Pour éviter que chaque exécution de votre programme produise la même séquence de nombres aléatoires, il est essentiel d'initialiser le générateur avec une valeur de départ différente. Cela se fait en utilisant la fonction `void srand(unsigned int seed);` (documentation ici). Une pratique courante consiste à utiliser l'heure actuelle comme graine, obtenue avec la fonction `time(NULL)` (documentation ici) :

```c
#include <stdlib.h>
#include <time.h>

int main() {
    srand(time(NULL));
    // Code generant des nombres aleatoires
    return 0;
}
```

Attention à l'utilisation de `rand()`

Évitez d'utiliser l'opérateur modulo pour générer un nombre dans une plage spécifique (par exemple, `rand() % n`) car cela introduit un biais si `RAND_MAX` + 1 n'est pas divisible par `n`.
Pour générer un nombre aléatoire entre `a` et `b` de manière uniforme, pensez à utiliser la division par `RAND_MAX + 1.0` et à ajuster le résultat pour qu'il soit dans l'intervalle souhaité. Réfléchissez à la manière dont vous pouvez utiliser la valeur retournée par `rand()` pour obtenir une distribution uniforme sur un intervalle spécifique.

Réflexion sur les intervalles larges

Lorsque l'intervalle entre `a` et `b` est plus grand que l'intervalle [0, `RAND_MAX`], il est nécessaire de combiner plusieurs appels à `rand()` pour obtenir une distribution uniforme. Ce problème est complexe et hors du cadre de ce TP, mais il est important d'en être conscient.

3.1. Complétez dans le fichier « `./src/random.c` » :

(a) La fonction `double random_double(double a, double b);` qui retourne un nombre aléatoire de type `double` compris entre `a` inclus et `b` exclu. Écrivez un code de test dans « `test_unit.c` ».

(b) La fonction `float random_float(float a, float b);` qui retourne un nombre aléatoire de type `float` compris entre `a` inclus et `b` exclu. Écrivez un code de test dans « `test_unit.c` ».

(c) La fonction `size_t random_size_t(size_t a, size_t b);` qui retourne un nombre aléatoire de type `size_t` compris entre `a` inclus et `b` exclu. Écrivez un code de test dans « `test_unit.c` ».

(d) La fonction `int random_int(int a, int b);` qui retourne un nombre aléatoire de type `int` compris entre `a` inclus et `b` exclu. Écrivez un code de test dans « `test_unit.c` ».

(e) La fonction `char random_char(char a, char b);` qui retourne un nombre aléatoire de type `unsigned char` compris entre `a` inclus et `b` exclu. Écrivez un code de test dans « `test_unit.c` ».

(f) La fonction `void random_init_string(char * c, size_t n);` qui remplit la chaîne de

caractères `c` de lettres majuscules aléatoires,`n` est la taille du tableau pointé par le pointeur `c`. Écrivez un code de test dans « `test_unit.c` ».

3.2. Assurez-vous que chaque fonction de génération développée ait des tests unitaires correspondants dans le fichier « `test_unit.c` ». Les tests unitaires sont cruciaux pour valider le bon fonctionnement de vos fonctions de génération aléatoire et pour vous aider à identifier rapidement les erreurs éventuelles. Voici quelques suggestions pour améliorer vos tests :

(a) Pour chaque fonction, vérifiez que les valeurs générées sont bien comprises dans l'intervalle souhaité. Par exemple, pour `random_double(a, b)`, assurez-vous que la valeur retournée est toujours `a <= valeur < b`.

(b) Testez la distribution des valeurs générées pour vous assurer qu'elles sont uniformément réparties. Par exemple, générez un grand nombre de valeurs et vérifiez leur répartition statistique.

Tester la distribution des valeurs générées

Pour vérifier que vos fonctions de génération de nombres aléatoires produisent des valeurs uniformément réparties, vous pouvez suivre les étapes suivantes :

1. **Génération d'un grand nombre de valeurs** :
   - Pour obtenir des résultats significatifs, générez un grand nombre de valeurs (par exemple, 10 000 ou plus) en appelant votre fonction de génération dans une boucle.
2. **Comptage des occurrences** :
   - Utilisez un tableau pour compter combien de fois chaque valeur possible est générée. Assurez-vous que votre intervalle est bien défini.
   - Par exemple, si vous testez la fonction `random_int(0, 10)`, utilisez un tableau de 10 éléments pour compter les occurrences de chaque valeur entre 0 et 9.
3. **Analyse des résultats** :
   - Après avoir généré et compté les valeurs, analysez la répartition des occurrences. Pour une distribution uniforme, chaque valeur devrait apparaître environ `N / (b - a)` fois, où `N` est le nombre total de valeurs générées.
   - Vous pouvez tolérer une certaine variation, mais les valeurs doivent être relativement proches les unes des autres.
4. **Interprétation des résultats** :
   - Si certaines valeurs apparaissent beaucoup plus souvent ou beaucoup moins souvent que d'autres, cela peut indiquer un problème avec votre fonction de génération.
   - Une bonne approche est de vérifier que chaque valeur est générée un nombre de fois dans une plage acceptable autour de la moyenne attendue.

En suivant ces étapes, vous pouvez vous assurer que votre générateur de nombres aléatoires produit une distribution uniforme et correcte.

(c) Pour `random_init_string`, vérifiez non seulement que les caractères générés sont des lettres majuscules, mais aussi que la chaîne de caractères est correctement terminée par un caractère nul (`\0`).

(d) Vérifiez le comportement des fonctions lorsque les bornes sont égales (`a == b`). La fonction doit retourner `a` dans ce cas.

4 Test de votre vecteur dynamique

Maintenant, vous allez créer un programme pour évaluer les performances de votre vecteur dynamique.

Ce programme prendra en paramètre le type de test à exécuter et le nombre de tests à exécuter, avec la structure suivante :

`./bench_vector test_type init_size n`

avec

- `test_type` le type du test à exécuter ;
- `init_size` la taille initiale du tableau dynamique ;
- `n` le nombre d'exécutions du test.

Pour cela, vous devrez utiliser les arguments `argc` et `argv` de la fonction `int main(int argc, char *argv[])`.

4.1. Complétez dans le fichier « `./src/bench_vector.c` » :

(a) Écrivez dans votre fonction `main` le code pour récupérer les arguments passés au programme et convertissez les arguments `init_size` et `n` en valeurs numériques de type `size_t`. (Pensez à utiliser la fonction `int sscanf(const char *str, const char *format, ...);` documentation ici.)

(b) Écrivez la fonction `void insert_erase_random(size_t init_size, size_t n);` qui alloue un `vector` de taille `init_size` et répète `n`-fois l'ajout et la suppression d'un élément à des positions aléatoires.

Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égal à `"insert_erase_random"`.

(c) Écrivez la fonction `void insert_erase_head(size_t init_size, size_t n);` qui alloue un `vector` de taille `init_size` et répète `n`-fois l'ajout et la suppression d'un élément en tête.

Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égal à `"insert_erase_head"`.

(d) Écrivez la fonction `void insert_erase_tail(size_t init_size, size_t n);` qui alloue un `vector` de taille `init_size` et répète `n`-fois l'ajout et la suppression d'un élément en queue.

Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égal à `"insert_erase_tail"`.

(e) Écrivez la fonction `void read_write_random(size_t init_size, size_t n);` qui alloue un `vector` de taille `init_size` et répète `n`-fois la lecture et réécriture de la valeur lue incrémentée de 1 à des positions aléatoires.

Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égal à `"read_write_random"`.

(f) Écrivez la fonction `void read_write_sequential(size_t init_size, size_t n);` qui alloue un `vector` de taille `init_size` et répète `n`-fois la lecture et réécriture de la valeur lue incrémentée de 1 avec un parcours de la tête vers la queue.

Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égal à `"read_write_sequential"`.

(g) Écrivez la fonction `void bubble_sort(size_t init_size, size_t n);` qui alloue un `vector` de taille `init_size` et répète `n`-fois l'écriture de tous les éléments du vecteur avec des valeurs aléatoires puis tri du vecteur avec le tri à bulles (documentation ici).

Puis ajoutez dans le `main`, le code pour appeler cette fonction quand le `test_type` est égal à `"bubble_sort"`.

4.2. Test des performances de votre structure pour les différents cas de test et différentes valeurs de `init_size` et `n`.

(a) Suivez le protocole expérimental suivant pour évaluer les performances de votre structure de données :

1. **Préparation des tests** :
   — Choisissez plusieurs valeurs pour `init_size` (par exemple, 100, 1000, 10000).
   — Choisissez plusieurs valeurs pour `n` (par exemple, 10, 100, 1000).

Explications sur le choix des valeurs de `n`

— La valeur de `n` permet de rendre la mesure du temps précise quand `init_size` est très petite, car le temps d'exécution sera très court. Dans ce cas, il faut utiliser de grandes valeurs de `n` pour obtenir des mesures significatives.
— Quand `init_size` augmente, vous pouvez diminuer la valeur de `n` car le temps d'exécution de chaque opération sera plus long.
— Une bonne méthode est de trouver, pour chaque valeur de `init_size`, une valeur de `n` permettant d'obtenir quelques secondes ou dizaines de secondes de mesure. Le temps unitaire peut alors être approximativement trouvé en divisant par `n`.
— Notez que les temps d'exécution ne sont pas strictement linéaires mais affine (temps d'initialisation dépendant principalement de `init_size` et temps d'exécution des opérations dépendant de `init_size` et de `n`).

2. **Mesure du temps d'exécution** :
   — Utilisez la commande `time` pour mesurer le temps d'exécution de votre programme pour chaque combinaison de `init_size` et `n`.
   — Exemple : `time ./bench_vector test_type init_size n`

Explications des valeurs retournées par `time`

La commande `time` retourne trois valeurs importantes :
— **real** : Temps réel écoulé entre le début et la fin de l'exécution du programme. Cela inclut tout le temps passé à attendre que d'autres processus s'exécutent.
— **user** : Temps CPU total consommé par le programme en mode utilisateur, c'est-à-dire le temps passé à exécuter le code du programme lui-même.
— **sys** : Temps CPU total consommé par le programme en mode noyau, c'est-à-dire le temps passé à exécuter les appels système pour le programme.

3. **Mesure des allocations et de la mémoire utilisée** :
   — Utilisez la commande `valgrind` pour mesurer le nombre d'allocations, de libérations et la quantité totale de mémoire allouée pour chaque test.
   — Exemple : `valgrind -tool=massif ./bench_vector test_type init_size n`
4. **Enregistrement des résultats** :

— Créez un fichier Excel ou Google Sheet pour enregistrer les résultats.
— Pour chaque combinaison de `init_size` et `n`, enregistrez le temps d'exécution, le nombre total d'allocations, le nombre total de libérations et la quantité de mémoire allouée.

5. **Analyse des résultats** :
   — Analysez les résultats pour chaque type de test (`insert_erase_random`, `insert_erase_hea` `insert_erase_tail`, `read_write_random`, `read_write_sequential`, `bubble_sort`).
   — Créez des graphiques dans votre fichier Excel ou Google Sheet pour visualiser l'impact de `init_size` et `n` sur le temps de calcul, le nombre d'allocations et la mémoire utilisée.

(b) Voici un exemple de tableau à utiliser dans Excel ou Google Sheet :

Test Type	**init_size**	**n**	**Temps (real)**	**Temps (user)**	**Temps (sys)**	**Mémoire (bytes)**
insert_erase_random	100	10	0.01	0.005	0.002	2048
insert_erase_random	100	100	0.10	0.05	0.02	20480
insert_erase_random	100	1000	1.00	0.50	0.20	204800
...	...	...	...	...	...	...

TABLE 1 – Tableau de résultats des tests de performances

(c) Instructions pour l'utilisation des commandes `time` et `valgrind` :
— **Commande** `time` :
   — Utilisez `/usr/bin/time -v ./bench_vector test_type init_size n` pour obtenir des informations détaillées sur le temps d'exécution et l'utilisation de la mémoire.
— **Commande** `valgrind` :
   — Utilisez `valgrind -tool=massif ./bench_vector test_type init_size n` pour obtenir des informations détaillées sur l'utilisation de la mémoire.
   — Utilisez `valgrind -tool=memcheck -leak-check=full ./bench_vector test_type init_size n` pour obtenir des informations sur les fuites de mémoire.