INF1015 - Programmation orientée objet avancée
Travail dirigé No. 5

14-Complexité et conteneur non contigu,
15-Bibliothèque de structures de données et algorithmes

Objectifs :	Permettre à l'étudiant de se familiariser avec les différents conteneurs, algorithmes et leur complexité
Durée :	1 séance de laboratoire
Remise du travail :	Avant 23h59 le dimanche 12 juin 2022
Travail préparatoire :	Le solutionnaire du TD4, et lecture de l'énoncé.
Documents à remettre :	Sur le site Github Classroom, vous remettrez l'ensemble des fichiers .cpp et .hpp en suivant la procédure de remise des TDs

---

Directives particulières

- Ce TD est une suite du TD4, il débute donc avec le solutionnaire du TD4.
- Vous pouvez ajouter d'autres fonctions/méthodes et structures/classes, pour améliorer la lisibilité et suivre le principe DRY (Don't Repeat Yourself).
- Il est interdit d'utiliser les variables globales; les constantes globales sont permises.
- Vous devez éliminer ou expliquer tout avertissement de « build » donné par le compilateur (avec /W4).
- Respecter le guide de codage, les points pertinents pour ce travail sont donnés en annexe à la fin.
- N'oubliez pas de mettre les entêtes de fichiers (guide point 33).

Dans le cours INF1007, nous vous avons introduit la notion de liste. En C++, cette notion est tout aussi présente, les listes étant des conteneurs de la librairie STL. Il est à noter que bien que la notion de vecteur et la notion de liste se ressemblent beaucoup, c'est-à-dire des éléments placés de façon séquentielle, il ne s'agit pas de la même structure. Les vecteurs ont la particularité d'être contigus en mémoire. Ceci offre certains avantages, par exemple l'accès par la position de l'élément en O(1), au détriment de certains inconvénients, par exemple l'ajout d'un élément entre deux cases copie tous les éléments qui suivent. Les listes quant à elles, ne sont pas contiguës en mémoire. Le principal avantage est la possibilité d'ajouter un nouvel élément entre deux éléments existant en O(1). De plus, puisqu'elles ne sont pas contiguës, elles peuvent supporter des tailles largement supérieures à celles possibles pour les vecteurs, car elles ne sont pas limitées par la fragmentation de la mémoire, surtout si chaque élément est assez gros (si les éléments sont petits, les pointeurs peuvent finalement prendre plus de place que les données « utiles »). Par contre, l'accès à un élément par sa position (son indice) se fait en O(n), car il faut parcourir toute la liste. Pour parcourir efficacement une liste et pouvoir se souvenir comment accéder rapidement à un élément, la notion d'itérateur a été inventée. En effet, grâce à un itérateur, le premier accès à un élément pourrait être de complexité O(n), mais les accès subséquents à ce même élément sera en O(1) car l'itérateur sera déjà sur celui-ci. Le but de ce travail n'est pas de réinventer la librairie STL, mais de se familiariser avec la notion de liste et d'itérateur en implémentant notre propre liste personnalisée. Tout comme dans les notes de cours, pour simplifier nous n'auront pas d'itérateur constant, donc pas de liste constantes.

On vous permet d'utiliser des pointeurs bruts propriétaires pour implémenter la liste, sans que ça soit visible de l'extérieur de l'implémentation, en utilisant gsl::owner pour indiquer quels pointeurs sont propriétaires (plus simple). Il est aussi possible de le faire avec des pointeurs intelligents. Attention de ne pas mettre des shared_ptr dans les deux directions de la liste double (problème de références circulaires, vu dans les notes de cours). Avec des unique_ptr, attention qu'ils ne peuvent pas être utilisés dans les deux directions.

Dans le TD précédent, vous aviez utilisé un vecteur pour y placer les héros et les vilains. Cette fois-ci, nous allons les placer dans notre liste liée personnalisée. Par conséquent, vous allez vous servir du solutionnaire du TD 4 fourni.

Travail à effectuer :

1. Liste liée et itérateur

1.1 Complétez la classe Nœud.
1.2 Complétez la classe Iterateur.
1.3 Complétez la classe ListeLiee.

2. Conteneurs

2.1 Utilisez un conteneur, pas un algorithme de tri (pas « sort », « qsort » …), pour avoir les héros en ordre alphabétique. Vérifiez que l'ordre est bon dans votre débogueur.
2.2 Affichez un des héros en le trouvant par son nom dans le conteneur en 2.1. Donnez en commentaire la complexité en moyenne de cette recherche (pas de l'affichage une fois trouvé). Expliquez pourquoi en commentaire.
2.3 Selon vous, lors d'une recherche d'un héros par le nom, quel conteneur entre la liste liée en 1. et celui utilisé en 2.1 permet la recherche la plus rapide. Expliquez pourquoi en commentaire.

ANNEXE 1 : Utilisation des outils de programmation et débogage

Utilisation des avertissements :
Avec les TD précédents vous devriez déjà savoir comment utiliser la liste des avertissements. Pour voir la liste des erreurs et avertissements, sélectionner le menu Affichage > Liste d'erreurs et s'assurer de sélectionner les avertissements. Une recompilation (menu Générer > Compiler, ou Ctrl+F7) est nécessaire pour mettre à jour la liste des avertissements de « build ». Pour être certain de voir tous les avertissements, on peut « Regénérer la solution » (menu Générer > Regénérer la solution, ou Ctrl+Alt+F7), qui recompile tous les fichiers.

Votre programme ne devrait avoir aucun avertissement de « build » (les avertissements d'IntelliSense sont acceptés). Pour tout avertissement restant (s'il y en a) vous devez ajouter un commentaire dans votre code, à l'endroit concerné, pour indiquer pourquoi l'avertissement peut être ignoré.

Rapport sur les fuites de mémoire et la corruption autour des blocs alloués :
Le programme inclus des versions de débogage de « new » et « delete », qui permettent de détecter si un bloc n'a jamais été désalloué, et afficher à la fin de l'exécution la ligne du programme qui a fait l'allocation. L'allocation de mémoire est aussi configurée pour vérifier la corruption lors des désallocations, permettant d'intercepter des écritures hors bornes d'un tableau alloué.

Utilisation de la liste des choses à faire :
Le code contient des commentaires « TODO » que Visual Studio reconnaît. Pour afficher la liste, allez dans le menu Affichage, sous-menu Autres fenêtres, cliquez sur Liste des tâches (le raccourci devrait être « Ctrl \ t », les touches \ et t faites une après l'autre). Vous pouvez double-cliquer sur les « TODO » pour aller à l'endroit où il se trouve dans le code. Vous pouvez ajouter vos propres TODO en commentaire pendant que vous programmez, et les enlever lorsque la fonctionnalité est terminée.

Utilisation du débogueur :
Lorsqu'on a un pointeur « ptr » vers un tableau, et qu'on demande au débogueur d'afficher « ptr », lorsqu'on clique sur le + pour afficher les valeurs pointées il n'affiche qu'une valeur puisqu'il ne sait pas que c'est un tableau. Si on veut qu'il affiche par exemple 10 éléments, il faut lui demander d'afficher « ptr,10 » plutôt que « ptr ».

Utilisation de l'outil de vérification de couverture de code :
Suivez le document « Doc Couverture de code » sur le site Moodle.

Annexe 2 : Points du guide de codage à respecter

Les points du **guide de codage** à respecter **impérativement** pour ce TD sont :
(voir le guide de codage sur le site Moodle du cours pour la description détaillée de chacun de ces points)

Mêmes points que le TD3 :

2 : noms des types en UpperCamelCase
3 : noms des variables en lowerCamelCase
5 : noms des fonctions/méthodes en lowerCamelCase
7 : noms des types génériques, une lettre majuscule ou nom référant à un concept
8 : préférer le mot typename dans les template
15 : nom de classe ne devrait pas être dans le nom des méthodes
21 : pluriel pour les tableaux (int nombres[];)
22 : préfixe *n* pour désigner un nombre d'objets (int nElements;)
24 : variables d'itération i, j, k mais jamais l, pour les indexes
27 : éviter les abréviations (les acronymes communs doivent être gardés en acronymes)
29 : éviter la négation dans les noms
33 : entête de fichier
42 : #include au début
44,69 : ordonner les parties d'une classe public, protected, private
46 : initialiser à la déclaration
47 : pas plus d'une signification par variable
48 : aucune variable globale (les constantes globales sont tout à fait permises)
50 : mettre le & près du type
51 : test de 0 explicite (if (nombre != 0))
52, 14 : variables vivantes le moins longtemps possible
53-54 : boucles for et while
58-61 : instructions conditionnelles
62 : pas de nombres magiques dans le code
67-78, 88 : indentation du code et commentaires
83-84 : aligner les variables lors des déclarations ainsi que les énoncés
85 : mieux écrire du code incompréhensible plutôt qu'y ajouter des commentaires