INF1015 - Programmation orientée objet avancée
Travail dirigé No. 3

6-Pointeurs intelligents, 7-Surcharge d'opérateurs,
8-Copie, 9-Template, 10-Lambda

Objectif :	Permettre à l'étudiant de se familiariser avec les unique/shared_ptr, la surcharge d'opérateurs, la copie d'objets avec composition non directe, les « template » et les fonctions d'ordre supérieur.
Durée :	Deux semaines de laboratoire.
Remise du travail :	Avant 23h59 le dimanche 29 mai 2022.
Travail préparatoire :	Solutionnaire TD2 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 refonte du TD2. Pour vous aider, la solution du TD2 est disponible sur Moodle.
- 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.
- Il est interdit d'utiliser std::vector, le but du TD est de faire l'allocation dynamique avec les pointeurs intelligents.
- 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).

ATTENTION : Afin de d'éviter les confusions entre développeur et designer, ce qu'on nommait des designers au TD2 sont maintenant des concepteurs.

Nous voulons recréer le TD2, mais en suivant les concepts de la POO cette fois-ci, pas seulement pour les deux classes qui étaient demandées d'ajouter. Comme vous l'avez sans doute remarqué, il y avait de la répétition de code au TD2, plus précisément une ListeJeux et une ListeDesigners. Pour respecter le principe DRY, nous allons fusionner ces deux classes similaires en une classe générique (*template*) nommée Liste, autrement dit, en une `Liste<T>`, et en particulier on voudrait que `ListeJeux` devienne `Liste<Jeu>`. De plus, puisque nous programmerons le tout en C++ moderne, nous allons donc laisser tomber les pointeurs bruts pour les remplacer par des pointeurs intelligents, `std::shared_ptr` et `std::unique_ptr` (aucun `new`); souvenez-vous qu'un pointeur partagé est un pointeur dont la possession peut être partagée par plusieurs objets. Finalement, afin de vous simplifier la tâche, les interfaces des classes à implémenter vous sont partiellement données. Il est possible que vous ayez à implémenter d'autres méthodes, opérateurs, constructeurs, destructeurs, etc. pour que ce qui est demandé fonctionne et suivre les bons principes. Attention aussi aux `const`. Notez que nous n'allons pas créer une liste de jeux participés pour les designers.
Vous pouvez écrire les méthodes qui ont une seule ligne dans la classe. Vous pouvez utiliser les `span` comme dans le TD2 mais ce n'est pas obligatoire.

Travail à faire :

1) Créer une classe générique nommée `Liste`. Cette classe doit se servir des pointeurs intelligents. C'est à vous de déterminer où ils doivent être utilisés. Les attributs de cette classe ressemblent beaucoup à ceux des structures `ListeJeux` et `ListeDesigners` du solutionnaire du TD2. On veut qu'un même élément puisse être dans plusieurs listes (le même élément, pas des copies de l'élément) et qu'il soit désalloué automatiquement quand il ne sera plus référé.

2) Dans la classe générique `Liste`, vous devrez implémenter une méthode pour pouvoir ajouter un élément à ladite `Liste`; tel que dit précédemment, on veut que cet élément puisse être un élément d'une autre liste, pour qu'il puisse être dans plusieurs listes, sans copier l'élément. Cette méthode ne devrait pas faire inutilement de comptage de références (pas de +1 suivi de -1 lors d'une utilisation simple de cette méthode).
3) Compléter ce qu'il faut pour que `creerListeJeux` retourne bien la liste des jeux lus du fichier; les fonctions sont similaires au TD2, mais devraient utiliser votre nouvelle classe `Liste`, ce qui permettra de vérifier qu'elle fonctionne par le fait même; à la fin de la lecture, le nombre de jeux dans la liste devrait être 17, et la capacité de la liste devrait être rendue à 32 (on double la capacité quand il manque de place). Vous pouvez aussi faire des tests plus simples avant, si vous voulez, par exemple avec une simple `Liste<int>`.
4) Toujours dans la classe générique `Liste`, on vous demande de surcharger l'opérateur `[]` pour pouvoir accéder aux éléments de la liste sans passer par un « getter » explicitement, ce qui pourrait être utile dans des boucles `for` qui itéreront sur lesdits éléments. Pour tester l'opérateur, et que la liste est bien créée, vérifiez dans le main que le titre du jeu à l'indice 2 est « Secret of Mana » et le nom de son concepteur à l'indice 1 est « Hiromichi Tanaka ».
5) Encore dans la classe générique `Liste`, implémenter une méthode pour chercher un élément en lui passant une fonction lambda pour indiquer le critère (similaire aux notes de cours). Elle devrait permettre n'importe quel critère et retourner le premier qui correspond. On veut pouvoir ajouter cet élément à une autre liste sans le copier. Puis dans la classe `Jeu`, implémenter une méthode pour chercher un concepteur en lui passant une lambda, qui utilise la méthode de recherche dans `Liste`. Pour la tester, chercher le concepteur « Yoshinori Kitase » dans les jeux aux indices 0 et 1, les deux devraient donner un pointeur vers la même adresse, et l'année de naissance devrait être 1966.
6) La classe `Liste` est une classe générique et on ne veut pas imposer une manière commune d'afficher à tous les types de listes. Par conséquent, pour afficher la liste de jeux ainsi que celle des concepteurs, nous allons surcharger l'opérateur d'insertion de flux `<<`. Le résultat souhaité est que l'instruction `std::cout << ligneSeparation << laListeDeJeux;` affiche au complet la liste de jeux, similairement au TD2. Puisque la liste contient des jeux et que les jeux contiennent des concepteurs, il vous sera nécessaire de surcharger cet opérateur plus d'une fois. **Faites la surcharge dans le main.cpp**. Vérifier que ça fonctionne aussi pour écrite dans un fichier texte (ofstream) au lieu de l'afficher à l'écran.
7) Tester la copie : Prendre le jeu à l'indice 2 et en faire une copie dans un nouveau `Jeu copieJeu = *lj[2]`. Remplacer le deuxième concepteur dans `copieJeu` par un autre venant du jeu à l'indice 0. Affichez le jeu à l'indice 2 et sa copie modifiée, pour voir que les liste de concepteurs sont différentes et vérifiez que l'adresse du premier `Concepteur` dans les deux listes est la même (donc que ce n'est pas une copie du concepteur).

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)

Nouveaux points pour le TD3 :

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
44,69 : ordonner les parties d'une classe public, protected, private

Points du TD2 :

2 : noms des types en `UpperCamelCase`
3 : noms des variables en `lowerCamelCase`
5 : noms des fonctions/méthodes en `lowerCamelCase`
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
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