INF1015 - Programmation orientée objet avancée
Travail dirigé No. 3

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

Objectifs :	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 23h30 le dimanche 5 mars 2023.
Travail préparatoire :	Avoir un TD2 fonctionnel, et lecture de l'énoncé.
Documents à remettre :	sur le site Moodle des travaux pratiques, vous remettrez l'ensemble des fichiers .cpp et .hpp compressés dans un fichier .zip en suivant la procédure de remise des TDs.

Directives particulières

- Ce TD est une suite du TD2, il reprend votre solution finale du TD2. Nous pourrons fournir un solutionnaire pour que vous puissiez corriger votre TD2 ou prendre notre solutionnaire du TD2 comme point de départ s'il est plus difficile de rendre fonctionnel votre code de TD2.
- 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 pointeur 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).

Ce TD est une série de modifications à apporter au TD2 en utilisant la matière des chapitres 6 à 10. On indique le chapitre principal dont la matière est requise pour effectuer les modifications, mais la matière d'autres chapitres vus peut aussi l'être.

Chapitre 6 :

1. Inclure <memory> et changer `Acteur**` en `unique_ptr<Acteur*[]>` dans `ListeActeurs`, et que le programme fonctionne encore. L'allocation du tableau, pour la bonne taille, devrait être faite dans un constructeur de `ListeActeurs`. Les méthodes que vous ajoutez à `ListeActeurs`, définissez leur corps directement dans la classe dans le .hpp, en prévision du changement en template demandé plus bas. (Pour simplifier l'écriture, on vous permet « `using namespace std;` » dans le .hpp, même si normalement on ne fait jamais ça. Les modules C++20 et la bibliothèque standard modulaire C++23 devraient régler le problème d'une manière plus propre.)
2. Donner des valeurs aux attributs dans les struct qui n'ont pas un constructeur s'assurant de donner des bonnes valeurs, pour que la construction par défaut donne des objets valides.
3. Commenter `joueDans` dans `Acteur`, la fonction `afficherFilmographieActeur` et son appel. Changer `unique_ptr<Acteur*[]>` en `unique_ptr<shared_ptr<Acteur>[]>`, et que le programme fonctionne encore. Le programme devrait fonctionner et les acteurs devraient être correctement désalloués au bon moment par `shared_ptr` (aucune fuite de mémoire), sans avoir à vérifier s'ils jouent encore dans des films (il n'est plus possible de le vérifier puisque `joueDans` n'existe plus).

Chapitre 7 :

Changer la fonction `afficherFilm` pour qu'on puisse afficher comme ceci:

```
cout << unFilm << unAutreFilm;
```

Et que ça fonctionne si on veut l'afficher sur autre chose que `cout`, par exemple dans un `ostringstream` (dans `<sstream>`) ou dans un fichier avec `ofstream` :

```
ostringstream tamponStringStream;
tamponStringStream << unFilm;
string filmEnString = tamponStringStream.str();

ofstream fichier("unfilm.txt");
fichier << unFilm;
```

Chapitres 7 – 8 :

Permettre les opérations suivantes dans le `main`, avant de détruire le premier film (le premier film est encore Alien et le deuxième Skyfall):

1. `Film skylien =` /* `listeFilms[0]` ou `*listeFilms[0]` selon ce qui fait du sens */;
2. Changer le titre du film `skylien` pour "Skylien".
3. Changer le premier acteur du film `skylien` pour le premier acteur de `listeFilms[1]`.
4. Changer le nom du premier acteur de `skylien` pour son nom complet "Daniel Wroughton Craig".
5. Afficher `skylien`, `listeFilms[0]` et `listeFilms[1]`, pour voir que Alien n'a pas été modifié, que `skylien` a bien l'acteur modifié et que `listeFilms[1]` a aussi l'acteur modifié puisque les films devraient partager le même acteur.

(ATTENTION à la construction du film dans `lireFilm`. Le code fourni dans le TD2 change directement les champs dans un `Film`, et à la fin de ces changements le `Film` est dans un état incorrect puisque `nElements > capacite`, et l'allocation du tableau n'a pas encore été faite. Une copie du `Film` à ce moment pourrait causer des problèmes...)

Chapitre 10 :

Ajouter une méthode à ListeFilms pour chercher un film en lui passant une lambda pour indiquer le critère, similairement aux notes de cours. Elle devrait permettre n'importe quel critère sur le film, et retourner le premier qui correspond. Utiliser cette méthode pour trouver le film dont la recette est 955M$; devrait donner le film Le Hobbit.

Chapitre 9 :

1. Convertir `ListeActeurs` en `class Liste` template pour pouvoir contenir autre chose que des `Acteur`, puis déclarer un équivalent (ligne à placer après la fin définition de la classe `Liste`):

   ```
   using ListeActeurs = Liste<Acteur>;
   ```

   (Le programme devrait encore fonctionner avec aucune modification autre que dans `ListeActeur` devenu `Liste`)
2. Pour vérifier que ça fonctionne, construisez une `Liste<string> listeTextes`, y mettre deux `string` quelconques, et copiez-la dans une nouvelle:

   `Liste<string> listeTextes2 = listeTextes;` // Devrait utiliser ce qui vous a permis de copier une ListeActeurs d'un Film pour supporter Film skylien = ... ci-dessus.
3. Remplacer `listesTextes[0]` par un nouveau texte (nouvelle allocation de `shared_ptr`), et modifiez le texte de `listesTextes[1]`.
4. Afficher les 4 textes, il devrait y avoir seulement 3 textes différents puisque `listesTextes[1]` et `listesTextes2[1]` réfèrent à un même texte partagé (la modification du texte de `listesTextes[1]` affecte les deux).

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