INF1015 - Programmation orientée objet avancée
Travail dirigé No. 2

Passage de paramètres
Allocation dynamique
Classes

Objectifs :	Permettre à l'étudiant de se familiariser avec l'allocation dynamique et aux classes; introduction au test avec couverture de code, et à l'utilisation d'un analyseur statique de code et débogueur.
Durée :	Deux semaines de laboratoire.
Remise du travail :	Avant 23h30 le dimanche 14 février 2021.
Travail préparatoire :	Lecture de l'énoncé, incluant l'annexe, et lire le document *Boucle sur intervalle, cppitertools et span* sur le site Moodle du cours.
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

- Les bibliothèques GSL et CPPItertools sont fournies et votre programme devrait avoir principalement des boucles sur intervalles plutôt que les anciens « for » ; vous aurez à modifier certains « for » qui sont déjà dans les fonctions où il y a des TODO. Le projet fourni est préconfiguré en C++latest (C++20).
- Vous pouvez ajouter d'autres fonctions 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 à la main.
- 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).

Exercice 1 : Allocation dynamique

Le fichier `films.bin` fourni, contient des informations sur plusieurs films ayant réalisé de grosses recettes (relativement à leur année de parution). En mémoire, nous voulons représenter une `ListeFilms`, chaque `Film` ayant plusieurs informations dont une `ListeActeurs`. Chaque liste (liste de films pour la collection et liste d'acteurs dans un film) est représentée par un nombre d'éléments et un pointeur vers un tableau de pointeurs. Nous voulons pouvoir ajouter/enlever des films de la liste sans faire une réallocation à chaque fois, donc les listes contiennent aussi une valeur pour la capacité. La capacité est le nombre de cases allouées dans le tableau (taille du tableau), alors que nElements indique le nombre de cases utilisées, aux positions 0 à nElements-1.

Voici des extraits des structures (voir le fichier `structures.hpp` pour les structures complètes) :

```
struct ListeFilms {
	int capacite, nElements;
	Film** elements;
};
struct ListeActeurs {
	int capacite, nElements;
	Acteur** elements;
};
struct Film {
	...
	ListeActeurs acteurs;
};
struct Acteur {
	...
	ListeFilms joueDans;
```

```
};
```

Noter que la seule différence entre les deux types de listes est le type de `elements`. Ici, `elements` est un pointeur vers un tableau de pointeurs, chaque pointeur du tableau étant vers un seul élément (soit un `Film` ou un `Acteur`); avec les `templates` qui sont seulement vus à la semaine 5 (donc après ce TD), il n'y aurait qu'une seule structure pour ces deux types de listes (on ne vous demande pas de le faire). La Figure 1 montre les différents pointeurs entre les structures. Le fait d'avoir un tableau de pointeurs plutôt qu'un tableau de `Film` (ou `Acteur`) permet de changer l'ordre des éléments et réallouer les tableaux sans copier les données elles-mêmes (uniquement en copiant les pointeurs), et permet d'avoir plusieurs listes qui réfères au même `Film` (ou `Acteur`) plutôt que d'avoir les mêmes informations plusieurs fois en mémoire (dans l'exemple de la Figure 1, on voit que le premier film en haut de la figure est pointé par la collection à sa gauche ainsi que par la liste de films dans lesquels joue l'acteur qui se trouve à droite sur la figure). Un `Acteur` contient une `ListeFilms` portant le nom `joueDans`, qui indique tous les films de la collection dans lesquels l'acteur joue.
Attention : Tel que montré sur la Figure 1, une `ListeFilms` contient un tableau dynamique de `Film`s, chaque `Film` contient une `ListeActeurs` qui contient un tableau dynamique d'`Acteur`s, et chaque `Acteur` contient une `ListeFilms`. On suppose qu'il n'y a qu'une seule collection principale (liste de films chargée du fichier) contenant tous les films, et que les listes de films dans les acteurs (les films dans lesquels ils jouent) pointent uniquement vers des films de la collection principale. Lorsqu'on désalloue un `Acteur`, le tableau de pointeurs de `joueDans` devra être désalloué, mais il ne faut pas désallouer les `Film`s puisqu'ils sont encore présents dans la collection (liste de films principale). Sur la figure, on fera donc la désallocation pour les flèches qui pointent vers la droite, mais pas les flèches qui sont vers la gauche.

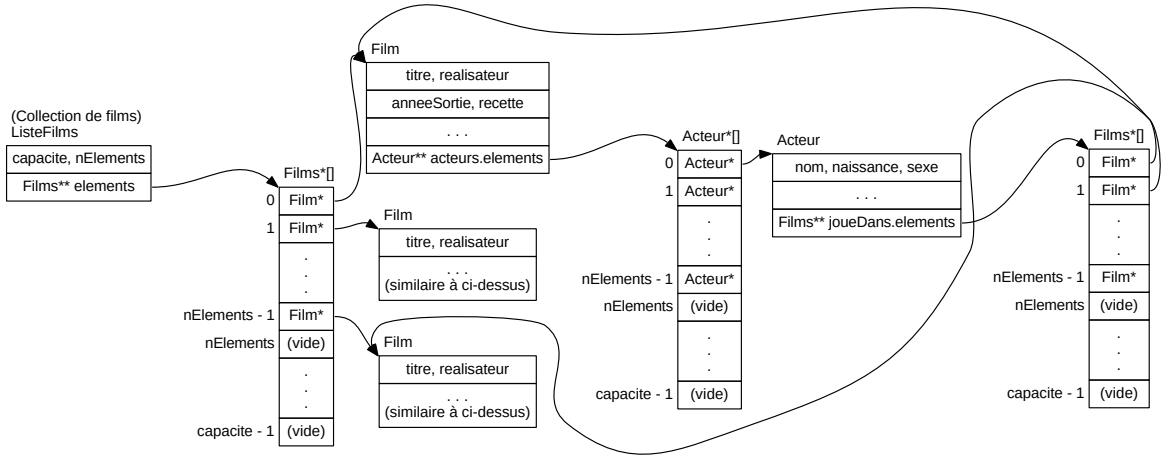

Figure 1. Pointeurs dans les structures de données.

La base de code fournie effectue déjà correctement la lecture du fichier, mais ne fait pas l'allocation de mémoire nécessaire pour conserver les données (le fichier est lu dans des variables locales qui sont immédiatement détruites après). Il n'est pas nécessaire de comprendre le format du fichier pour faire le TD (il n'est pas fait pour être lisible dans un éditeur texte).

Partie 1 :

Suivez les commentaires `//TODO:` dans le squelette de programme fourni (uniquement dans le fichier `td2.cpp`). Vous pouvez/devez ajouter les paramètres requis aux fonctions, donc si un commentaire indique qu'une fonction doit avoir un certain paramètre, elle devrait normalement avoir ce paramètre, mais il est probable qu'elle aura aussi besoin d'autres paramètres qui n'ont pas été explicitement dits.

De manière générale, il faut :

- Fonction pour **ajouter un film à une liste**, qui fait la réallocation du tableau en doublant sa capacité s'il ne reste pas de place en s'assurant qu'il y a au moins une capacité d'un élément (sinon, le double de zéro resterait zéro). En C++ il n'est pas possible de changer la taille d'une allocation, alors il faut allouer un nouveau tableau, copier de l'ancien au nouveau et détruire l'ancien tableau trop petit. Le film à ajouter est déjà alloué, il faut simplement ajouter à la liste le pointeur vers le film existant.
- Fonction pour **enlever un film d'une liste**, qui prend un pointeur vers un film et enlève ce film de la liste sans détruire le film. L'ajout du film utilisait un film existant, et le film existe encore après avoir enlevé le film de la collection, ces fonctions sont donc symétriques. Des fonctions séparées serviront à créer et détruire les films.
- Fonction pour **trouver un acteur**, qui cherche dans tous les films d'une collection un acteur par son nom, et retourne un pointeur vers cet acteur (ou nullptr si l'acteur n'est pas trouvé). On suppose que le nom d'un acteur l'identifie de manière unique, i.e. si on voit le même nom deux fois, c'est le même acteur.
- Fonctions pour **créer une collection** à partir du fichier (`creerListe`/`lireFilm`/`lireActeur`), qui allouent la capacité nécessaire pour les films dans le fichier, qui charge les données de chacun de ces films; chaque film contient une liste d'acteurs, qu'il faut aussi allouer, et il faut allouer la mémoire pour chaque acteur. Attention : Le fichier contient certains acteurs plus d'une fois, mais nous voulons qu'en mémoire l'allocation soit faite une seule fois par acteur différent (on utilisera la fonction pour trouver un acteur par nom, pour vérifier si un acteur a déjà été alloué).
- Fonction pour **détruire un film**, qui libère toute la mémoire liée au film, incluant le tableau dynamique d'acteurs. Attention : La mémoire liée à un acteur doit être aussi libérée, mais uniquement si l'acteur ne joue pas dans d'autres films. Lors de la destruction d'un film, il faut donc enlever ce film de la liste des films dans lesquels l'acteur joue, et désallouer l'acteur s'il n'est plus dans aucun film (si sa liste `joueDans` est rendue vide).
- Fonction pour **détruire une collection complète**, utilisant la fonction de destruction ci-dessus.
- Fonctions pour **afficher un seul film**, et pour **afficher tous les films d'une collection**.
- Fonction pour **afficher les films dans lesquels un acteur joue**.

Partie 2 :

Convertir la `struct ListeFilms` en une classe qui suit les principes de programmation orientée objet. Les attributs devraient être privés, les fonctions sur ces listes devraient devenir des méthodes de la classe, la construction de l'objet devrait toujours donner un objet valide. L'utilisation des paramètres et méthodes « const » doit être adéquate. Vos méthodes ne devraient pas permettre de modifier l'objet de manière incohérente (p.ex. mettre une capacité qui ne correspond pas à la taille allouée du tableau). Noter que nous verrons comment correctement copier ou déplacer un tel objet seulement à la semaine 4, et pour cette raison nous vous proposons de faire la désallocation dans une méthode plutôt que dans le destructeur (sinon `creerListe` va désallouer la liste lors de la destruction de ses variables locales et c'est la liste désallouée qui sera utilisée dans le `main`).

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)

Points du TD6 :

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