INF1005C - PROGRAMMATION PROCÉDURALE
Travail dirigé No. 6
Représentation des données
Allocation dynamique
Test et débogage

Objectifs :	Permettre à l'étudiant de se familiariser avec l'allocation dynamique et la représentation des données; introduction au test avec couverture de code, et à l'utilisation d'un analyseur statique de code et débogueur.
Durée :	Deux séances de laboratoire.
Remise du travail :	Avant 23h30 le mardi 4 décembre 2018.
Travail préparatoire :	Lecture de l'énoncé, incluant l'annexe, et rédaction des algorithmes.
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

- N'oubliez pas les entêtes de fichiers ni, aux endroits appropriés, les commentaires dans le code.
- Vous devez utiliser les caractères accentués et ou symboles spéciaux dans les affichages, mais vous n'avez pas à les supporter en entrée.
- Vous devez éliminer ou expliquer tout avertissement donné par le compilateur (avec /W4) et l'analyseur (Cppcheck).
- Vous devez **obligatoirement** recourir au débogueur pour la mise au point de votre programme. Le chargé de lab peut vous demander d'afficher le contenu d'une variable dans le débogueur, vous devriez être capable.
- Suivez en tout temps le guide de codage (voir l'annexe 2).

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, comme dans le TD5, donc les listes contiennent aussi une valeur pour la capacité. 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`); si les `templates` étaient matière au cours, il n'y aurait qu'une seule structure pour ces deux types de listes. 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).

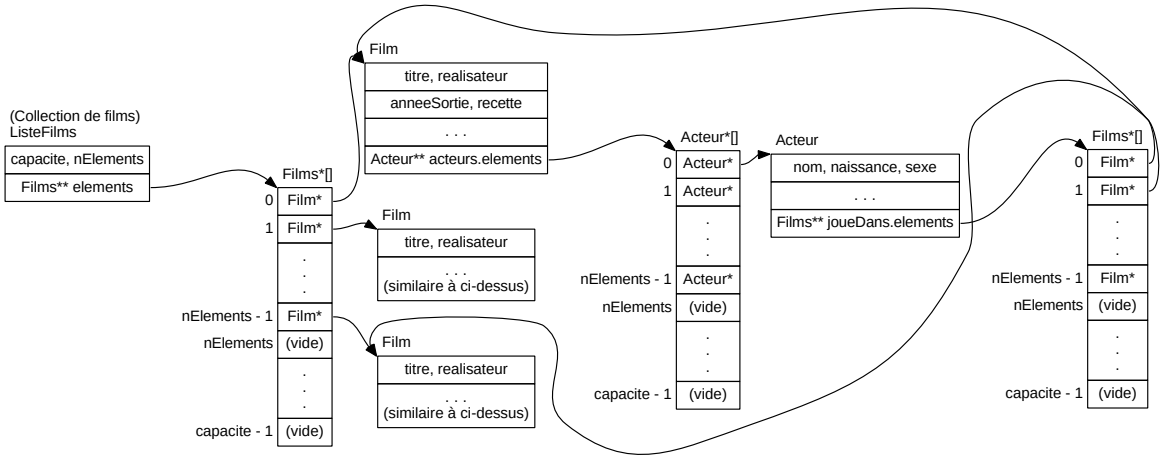

Figure 1. Pointeurs dans les structures de données.

La base de code fournie effectue déjà correctement la lecture du fichier sauf le choix des types à utiliser pour avoir les bonnes tailles d'entiers, 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). Pour réduire la taille du fichier, nous utilisons des entiers sur moins d'octets que l'entier standard. Vous devez premièrement définir correctement les types `UInt8` et `UInt16` (voir les `typedef` dans le fichier `Exercice1.cpp`), pour que ce soit des entiers non-signés sur 8 et 16 bits respectivement (si ces types sont mal définis, la lecture du fichier plantera fort probablement). La description du format du fichier se trouve à la fin de l'énoncé de l'exercice.

Suivez les commentaires `//TODO:` dans le squelette de programme fourni (uniquement dans le fichier `Exercice1.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 (inspirez-vous de votre fonction du TD5) en s'assurant qu'il y a au moins une capacité d'un élément (sinon, le double de zéro resterait zéro). 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**.

Format du fichier binaire (les fonctions fournies lisent déjà correctement le bon format si vous définissez correctement UInt8 et UInt16) :

Nous utilisons ici une syntaxe de style C/C++ pour décrire le format du fichier, mais il contient des tableaux dont la taille est connue uniquement à l'exécution, donc pas du C/C++ valide. Les éléments sont de taille variable, pour réduire la taille du fichier, et il n'est donc pas possible de faire un simple `read` pour lire d'un coup la structure comme dans le TD5.

Format de fichier {
```
  UInt16 nFilms;  Film films[nFilms];
```
}
Format `Film` dans le fichier {
```
  Chaîne titre, réalisateur;
  UInt16 annéeSortie, recette;
  UInt8  nActeurs;  Acteur acteurs[nActeurs];
```
}
Format `Acteur` dans le fichier {
```
  Chaîne nom;  UInt16 annéeNaissance;  char sexe;
```
}
Format `Chaîne` dans le fichier {
```
  UInt16 longueur;  wchar_t texte[longueur];
```
}

Par exemple, un acteur ressemblerait à ceci dans le fichier (en base 16) :
```
08 00 4A 00 6F 00 68 00 6E 00 20 00 44 00 6F 00 65 00 8A 07 4D
\ 8 / \ J / \ o / \ h / \ n / \   / \ D / \ o / \ e / \1930/ M
```

L'entier 16 bits ayant la valeur 8 indique le nombre de caractères dans le nom, il y a ensuite ce nombre de wchar_t (des caractères sur 2 octets, pour Unicode), ensuite encore un entier 16 bits (ici, la valeur 1930 en base dix est 078A en base seize), puis le caractère pour le sexe (ici, 'M' représenté en ASCII comme 4D en base seize).

Exercice 2 : Conversion de nombres

Vous devez implémenter les trois fonctions **`dec2int, hex2int`** et **`int2dec`**, dont vous devez décider des paramètres d'entrée et de retour, puis écrivez un programme principal qui demande à l'usager un nombre décimal, un hexadécimal, et qui affiche la somme des deux comme un nombre décimal.

- **`dec2int`** transforme le texte d'un nombre décimal en une valeur sur laquelle on peut faire des opérations arithmétiques en C++;
- **`hex2int`** transforme le texte d'un nombre hexadécimal en une valeur …;
- **`int2dec`** transforme une valeur du même type que le résultat des deux fonctions ci-dessus en sa notation hexadécimale textuelle.

Votre programme final ne doit pas utiliser cout, cin, les stringstream, atoi, itoa, to_string, stoi, ou toute autre fonction prédéfinie qui pourrait faire la conversion à votre place. Pour l'affichage et la lecture du clavier, utilisez uniquement les deux fonctions fournies « afficherTexte » et « lireEntree ».

ANNEXE 1 : Utilisation des outils de programmation et débogage.

Utilisation d'Unicode :
Pour afficher des caractères accentués, vous devez utiliser les versions « wide » de cout, string, et char, qui sont respectivement wcout, wstring, et wchar_t. Les chaînes et caractères « wide » s'écrivent avec un L majuscule devant, tel que L"Allô" et L'ô'. Des chaînes et caractères ordinaires peuvent aussi être affichés sur wcout mais ne doivent pas contenir de caractères accentués. Un exemple d'affichage se trouve dans le programme fourni.

Utilisation des avertissements et de Cppcheck :
Le but de ces outils est de vous aider à trouver certaines erreurs dans votre programme. Dans les propriétés du projet, dans C/C++, il est possible de choisir le niveau d'avertissements. La solution VS qu'on vous fournit est déjà configurée avec Cppcheck et le niveau d'avertissement /W4. Ceci permet de détecter certains problèmes possibles, tel que l'utilisation d'une affectation au lieu d'une égalité dans une condition (comme dans « if (x = 4) » qui devait probablement être « if (x == 4) »), les variables potentiellement non initialisées, et les conditions constantes.

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. 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, ni par le compilateur, ni par Cppcheck. 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 à chaque allocation, permettant d'intercepter des écritures hors bornes d'un tableau alloué. Si une corruption est détectée, le débogueur affichera un message indiquant une possible « défaillance du tas », et la fenêtre « Pile des appels » vous permettra de voir à quel endroit la détection a été faite (regardez vers le bas de la pile d'appels pour l'élément le plus haut qui correspond à votre programme, et non ceux au dessus qui correspondent aux fonctions de C++). Si vous avez un problème de corruption, vous pouvez utiliser « `_CrtCheckMemory()` » à n'importe quelle ligne pour vérifier si la corruption arrive avant ou après cette ligne (la fonction retourne faux s'il y a corruption). Il est aussi possible d'exécuter la vérification dans le débogueur; dans la liste des espions (fenêtre « Espion 1 »), ajoutez l'espion « `((int(*)())ucrtbased.dll!_CrtCheckMemory)()` ». La valeur sera 0 s'il y a corruption, 1 sinon, et il suffit de cliquer sur les flèches en rond pour revérifier la corruption à n'importe quel moment pendant qu'on trace dans le débogueur.

Utilisation de la liste des choses à faire :
Le code contient des commentaires « TODO » que Visual Studio reconnaît. Vous devez premièrement activer l'option, dans le menu Outils > Options..., allez dans Éditeur de texte > C/C++ > Mise en forme > Divers > Énumérer les tâches de commentaire, pour mettre cette option à « True », puis OK. Pour afficher la liste, allez dans le menu Affichage, sous-menu Autres fenêtres, cliquez sur Liste des tâches. Choisissez ensuite dans cette fenêtre d'utiliser les commentaires. 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 :
Vous devez obligatoirement utiliser l'outil de débogage pour regarder les valeurs et les pointeurs durant l'exécution de votre programme. Lorsqu'on a un pointeur « ptr » vers un tableau, et qu'on demande à l'outil 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 les mêmes que pour le TD précédent : (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 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`
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
89 : entêtes de fonctions; y indiquer clairement les paramètres [out] et [in,out]