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 :	Une semaine de laboratoire.
Remise du travail :	Avant 23h30 le dimanche 22 mai 2022
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 `jeux.bin` qui vous est fourni contient des informations sur plusieurs jeux vidéos (console et PC) qui ont connu un grand succès et dont certains ont débuté une série qui se poursuit encore de nos jours. En mémoire, nous voulons représenter une `ListeJeux`, chaque `Jeu` ayant plusieurs informations dont une `ListeDesigners`. Chaque liste (la liste de jeux pour la collection de jeux vidéo et la liste de designers d'un jeu) est représentée par une structure contenant un champ pour le nombre d'éléments et un pointeur vers un tableau de pointeurs. Nous voulons pouvoir ajouter/enlever des éléments d'une liste sans faire une réallocation à chaque fois, donc la liste contiendra aussi une valeur pour la capacité du tableau dynamique. La capacité est le nombre de cases allouées dans le tableau (taille du tableau), alors que nombre d'éléments indique le nombre de cases utilisées, et ce à partir de la case 0.

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

```
struct ListeJeux
{      unsigned int nElements, capacite;
       Jeu** elements;
};
struct ListeDesigners
{
       unsigned int nElements, capacite;
       Designers** elements;
};
```

```
struct Jeu
{
	...
	ListeDesigners designers;
};
struct Designer
{
	...
	ListeJeux listeJeuxDesignes;
};
```

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 (donc deux * au lieu d'une seule), chaque pointeur du tableau est le contenu d'un pointeur d'un seul élément (soit un `Jeu` ou un `Designer`). La Figure 1 montre les différents pointeurs entre les structures. Le fait d'avoir un tableau de pointeurs plutôt qu'un tableau de `Jeu` (ou `Designer`) 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 réfèrent au même `Jeu` (ou `Designer`) plutôt que d'avoir les mêmes informations plusieurs fois en mémoire (dans l'exemple de la Figure 1, on voit que le premier jeu vidéo en haut de la figure est pointé par la collection à sa gauche ainsi que par la liste de jeux dans lesquels le designer a contribué qui se trouve à droite sur la figure). Un `Designer` contient une `ListeJeux` portant le nom `listeJeuxParticipes`, qui indique tous les jeux de la collection auxquels le designer a contribué.

Attention : Tel que montré sur la Figure 1, une `ListeJeux` contient un tableau dynamique de jeux, chaque `Jeu` contient une `ListeDesigners` qui contient un tableau dynamique de designers, et chaque `Designer` contient une `ListeJeux`. On suppose qu'il n'y a qu'une seule collection principale (liste de jeux chargée du fichier) contenant tous les jeux, et que les listes de jeux des designers pointent uniquement vers des jeux de la collection principale. Lorsqu'on désalloue un `Designer`, le tableau de pointeurs de `listeJeuxParticipes` devra être désalloué, mais il ne faut pas désallouer les jeux puisqu'ils sont encore présents dans la collection de jeux principales. 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.

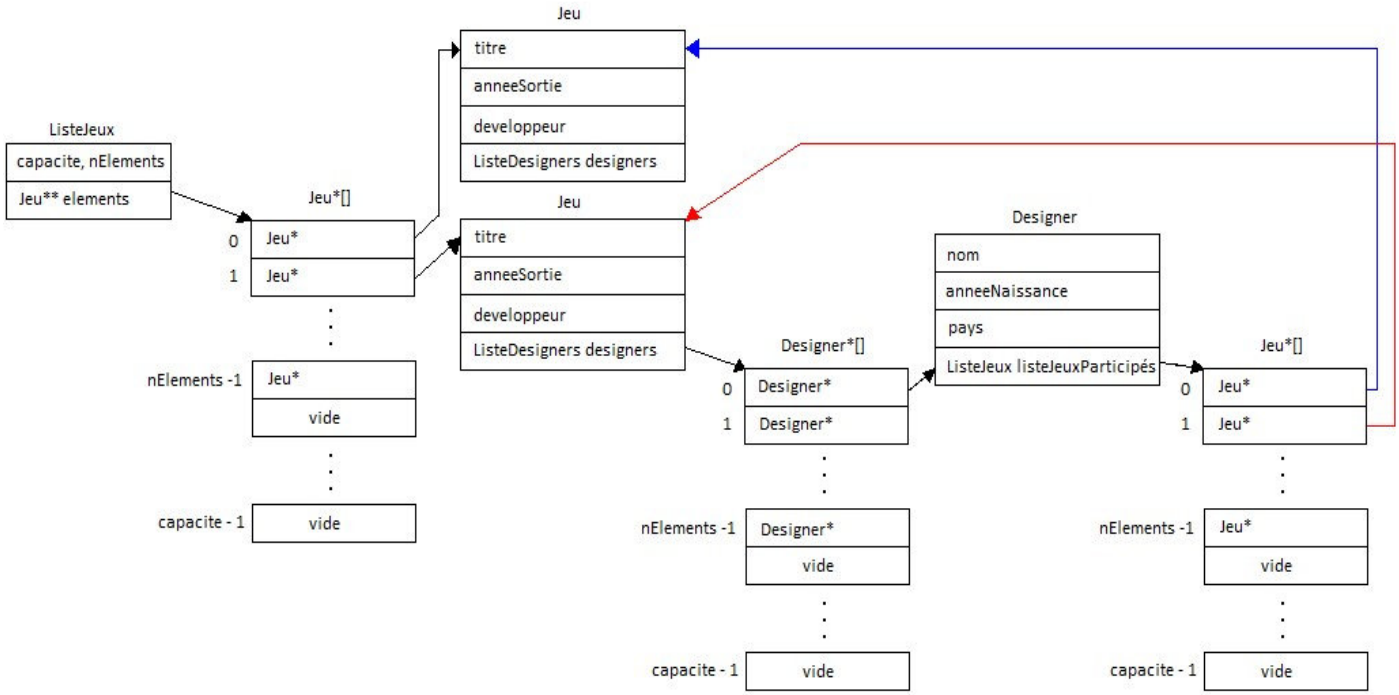

Figure 1. Pointeurs dans les structures de données.

La base de code fourni 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). Vous pouvez mettre tous les fichiers du TD dans le répertoire principal du « Projet VS/VSCode "vide" » fourni sur le site Moodle puis les ajouter dans le projet. Le code fourni devrait compiler et afficher des noms de jeux et designers.

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 jeu 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 jeu à ajouter est déjà alloué, il faut simplement ajouter à la liste le pointeur vers le jeu existant.
- Fonction pour **enlever un jeu d'une liste**, qui prend un pointeur vers un jeu et enlève ce jeu de la liste sans détruire le jeu. L'ajout du jeu utilisait un jeu existant, et le jeu existe encore après avoir enlevé le jeu de la collection, ces fonctions sont donc symétriques. Des fonctions séparées serviront à créer et détruire les jeux.
- Fonction pour **trouver un designer**, qui cherche dans tous les jeux d'une collection un designer par son nom, et retourne un pointeur vers ce designer (ou nullptr si le designer n'est pas trouvé). On suppose que le nom d'un designer l'identifie de manière unique, i.e. si on voit le même nom deux fois, c'est le même designer.
- Fonctions pour **créer une collection** à partir du fichier (`creerListe`/`lireJeu`/`lireDesigner`), qui allouent la capacité nécessaire pour les jeux dans le fichier, qui charge les données de chacun de ces jeux; chaque jeu contient une liste de designers, qu'il faut aussi allouer, et il faut allouer la mémoire pour chaque designer. Attention : Le fichier contient certains designers plus d'une fois, mais nous voulons qu'en mémoire l'allocation soit faite une seule fois par designer différent (on utilisera une fonction pour trouver un designer par nom, pour vérifier si un designer a déjà été alloué).
- Fonction pour **détruire un jeu**, qui libère toute la mémoire liée au jeu, incluant le tableau dynamique de designers. Attention : La mémoire liée à un designer doit être aussi libérée, mais uniquement si le designer ne participe plus à aucun jeu. Lors de la destruction d'un jeu, il faut donc enlever ce jeu de la liste des jeux dans lesquels le designer participe, et désallouer le designer s'il n'est plus dans aucun jeu (si sa liste `listeJeuxParticipes` est rendue vide).
- Fonction pour **détruire une collection complète**, utilisant la fonction de destruction ci-dessus.
- Fonctions pour **afficher un seul jeu**, et pour **afficher tous les jeux d'une collection**.

Partie 2 :

Les structures `Developpeur` et `ListeDeveloppeurs` doivent maintenant être des classes qui suivent les principes de programmation orientée objet. Les attributs devraient être privés, la construction de l'objet devrait toujours donner un objet valide, vos méthodes publiques 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). L'utilisation des paramètres et méthodes « const » doit être adéquate. On vous fournit la définition des attributs des classes et il faut ajouter les méthodes. Il faut tester vos méthodes dans le fichier main.cpp.

Classe `Developpeur`

Le constructeur permet d'initialiser l'attribut `paireNomJeux_`. Cet attribut est une paire dont le premier élément est le nom du développeur et le second est la liste des jeux développés par le développeur. À la création d'une instance, on initialise le nom du développeur (passé au constructeur) et la liste des jeux est vide. De manière symétrique, la destruction de l'objet doit détruire la liste de jeux sans détruire les jeux.

On veut pouvoir obtenir le nom du développeur.

On veut pouvoir compter le nombre de jeux qu'un développeur a fait dans une liste. La méthode s'applique sur un développeur et reçoit la liste des jeux dans laquelle compter le nombre de fois où on trouve ce développeur.

On veut pouvoir mettre à jour la liste de jeux d'un développeur en lui passant l'ensemble de la liste de tous les jeux (pas seulement les jeux développés par ce développeur). Cette fonction utilise le comptage de jeux indiqué ci-dessus pour ne pas avoir à faire plusieurs réallocations pendant que les jeux développés par ce développeur sont ajoutés à sa liste (qui se trouve dans second élément de l'attribut `paireNomJeux_`).

On veut pouvoir afficher les titres de tous les jeux développés par un développeur.

Classe `ListeDeveloppeurs`

Le constructeur crée une liste valide contenant aucun élément. Le destructeur libère le tableau dynamique.

La méthode `afficher()` parcourt le tableau et affiche son contenu en faisant appel à la méthode correspondante de la classe `Developpeur`.

La méthode `ajouterDeveloppeur` ajoute un `Developpeur` existant (passé par adresse; cette méthode n'alloue pas de `Developpeur`) uniquement s'il n'existe pas dans le tableau dynamique. Si la capacité du tableau est atteinte on augmente la capacité du tableau dynamique (utiliser la même façon que l'allocation dynamique dans la partie 1 en copiant le code dans une/des méthodes appropriées).

La méthode `retirerDeveloppeur` retire un `Developpeur` existant (passé par adresse) du tableau dynamique.

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 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