INF1015 - Programmation orientée objet avancée
Travail dirigé No. 4

11-Méthodes virtuelles et classes abstraites, 12-Conversion d'objets, 13-Héritage multiple

Objectifs :	Permettre à l'étudiant de se familiariser avec les méthodes virtuelles, les objets polymorphes, l'héritage simple et multiple
Durée :	Deux séances de laboratoire
Remise du travail :	Avant 23h30 le mardi 8 novembre 2022
Travail préparatoire :	Téléchargement des fichiers fournis 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 continue avec le thème des jeux vidéos. Contrairement aux TDs précédents cependant, presque aucun code ne vous est fourni. Vous devez donc coder de A à Z votre solution. Par conséquent, les fonctions, classes et méthodes à implémenter seront très simples, l'emphase étant mise sur votre compréhension des concepts.
- Vous devez vérifier que vos fonctions fonctionnent à mesure que vous les écrivez.
- Vous aurez fort probablement à ajouter d'autres fonctions/méthodes et/ou structures/classes, pour améliorer la lisibilité et suivre le principe DRY (Don't Repeat Yourself).
- Chaque classe décrite dans l'énoncé devrait avoir son .hpp qui correspond.
- Pour alléger l'écriture, on permet les « using namespace » dans tous les fichiers (même si c'est normalement très contre-indiqué de le faire globalement dans un .hpp).
- Il est interdit d'utiliser les variables globales; les constantes globales sont permises.
- Vous pouvez maintenant utiliser std::vector.
- Vous devez éliminer ou expliquer tout avertissement de « build » donné par le compilateur (avec /W4).
- Respectez 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).

`std::vector`

Maintenant qu'on sait faire l'allocation dynamique... vous pouvez utiliser `std::vector`, qui s'occupe de la réallocation du tableau. `vecteur.push_back(element)` est l'équivalent C++ de `liste.append(element)` en Python; on peut aussi utiliser `vecteur.resize(taille)` pour changer la taille (met des objets initialisés avec le constructeur par défaut dans les cases supplémentaires); le pointeur brut du début du tableau est `vecteur.data()`. La documentation sur cppreference.com est toujours utile.

Attributs privés : Dans ce TD, pour vous faire pratiquer les attributs privés avec accesseurs, et vous inciter fortement à mettre les accès aux attributs dans les bonnes classes, on vous demande que les attributs des classes demandées dans l'énoncé soient privés. Vous n'avez pas à écrire les accesseurs qui ne sont pas nécessaires pour le TD, et il devrait y en avoir peu à écrire; si vous avez à écrire des accesseurs pour tous les attributs, vous n'avez pas bien compris où les accès aux attributs devraient être faits.

Travail à effectuer :

Un jeu vidéo bien conçu implique généralement un héros, des alliés et un vilain ayant un objectif mettant en péril soit le héros, ses alliés, le monde, etc. C'est donc pour cette raison que vous aurez à lire deux fichiers binaires : heros.bin et vilains.bin. Le premier regroupe les héros de plusieurs jeux et le deuxième les vilains des jeux respectifs. Vous allez considérer les héros et les vilains comme étant à priori des personnages.

Vos constructeurs doivent utiliser uniquement les listes d'initialisation, si possible, plutôt que des affectations dans leur corps.
Comme le dit les notes de cours, « Les « cast » moins il y en a, mieux c'est »; on peut écrire ce programme sans aucun dynamic_cast ou static_cast.

1. Créer une classe complètement abstraite (interface) nommée `Affichable` ayant deux méthodes virtuelles pures nommées `afficher` et `changerCouleur`. L'« affichage » doit pouvoir se faire sur `cout`, dans un fichier, ou autre flux de sortie. Pour `changerCouleur`, un deuxième paramètre permet de spécifier une couleur avec laquelle nous voulons afficher les informations à la console (les mêmes codes de couleurs seront aussi envoyés dans les autres types de flux de sortie). Ce paramètre peut être un entier, un caractère ou une chaîne de caractères selon votre choix. Ce paramètre décidera de la couleur pour les affichages suivants. Pour changer la couleur du texte à la console, il faut se servir des séquences d'échappement. Celles-ci sont des chaînes de caractères que l'on passe à la console (`std::cout`). Exemple, la chaîne "`\033[91m`" passe la couleur rouge brillant à la console pour l'affichage des caractères. La couleur est représentée par le numéro 91, d'autres couleurs sont disponibles, voir cette page Wikipedia. Si on remplace le chiffre 91 par 0, la console est rétablie à son état d'origine. C'était de cette façon que nous avions pu dans les TDs précédents colorier la ligne de séparation.

2. Un `Personnage` a les attributs nécessaires pour y placer le nom d'un personnage (héros ou vilain) ainsi que le titre du jeu dans lequel ce personnage fait sa première apparition. On veut qu'un personnage soit Affichable dans un format lisible et bien étiqueté, pour présenter ses attributs.

   Exemple d'affichage :
   Nom : Randi
   Parution : Secret of Mana

3. Un `Heros` est un Personnage mais contient aussi une liste des noms de chacun de ses alliés ainsi que le nom de l'ennemi qu'il doit vaincre. Son affichage doit présenter les informations du personnage suivies de l'information supplémentaire le décrivant; vous devez évidemment éviter la répétition de code.

   Exemple d'affichage :
   Nom : Randi
   Parution : Secret of Mana
   Ennemi : Thanatos
   Alliés :
   Primm
   Popoi

4. Un `Vilain` est aussi un Personnage et il a un objectif (description textuelle), et son affichage doit présenter cette information en plus de celle du personnage.

   Exemple d'affichage :
   Nom : Thanatos
   Parution : Secret of Mana
   Objectif : Contrôler la forteresse de Mana afin de diriger le monde

5. Un `VilainHeros` est un `Vilain` et un `Heros`, sans avoir une double personnalité : il n'a qu'un nom mais il sera composé. Le VilainHeros conserve ce qu'il a de spécifique au vilain et héros mais a en plus une mission spéciale. Sa construction se fait à partir d'un vilain et d'un héros, et son nom sera la concaténation du nom du vilain avec celui du héros, séparés par un trait d'union. Le jeu de parution sera aussi la concaténation du jeu du vilain avec celui du héros, séparés par un trait d'union. La mission spéciale du VilainHeros est l'objectif du vilain dans le monde du héros (jeu de parution du héros).

Exemple d'affichage pour la fusion du vilain Kefka avec le héros Crono :
Nom : Kefka-Crono
Parution : Final Fantasy IV (III)-Chrono Trigger
Objectif : Détruire tout ce qui existe
Ennemi : Lavos
Alliés :
- Lucca
- Marle
- Frog
- Robo
- Ayla
- Magus

Mission spéciale : Détruire tout ce qui existe dans le monde de Chrono Trigger
(La couleur ci-dessus est pour indiquer d'où provient l'information, vous n'avez pas à faire l'affichage multi-colore.
Le texte en rouge étant de l'information provenant de la classe Vilain
Le texte en bleu étant de l'information provenant de la classe Heros)

6. Pour votre `main`,

- Créer d'abord trois vecteurs : un de type `Heros`, un autre de type `Vilain` et le dernier pour y mettre des `Personnages`.

- Faire la lecture des deux fichiers binaires afin de placer les héros et les vilains dans les bons vecteurs. Voir l'annexe 1 pour la structure des fichiers binaires.

- Ensuite, afficher tous vos héros séparés par une ligne et tous vos vilains séparés par une ligne avec des boucles `for` en faisant appel à la méthode `afficher`, de manière à ce que l'information s'affiche à la console.

- Afficher les informations des héros d'une couleur différente de celles des vilains. Par exemple en bleu pour les héros, en rouge pour les vilains.

- Placer tous les héros et vilains dans le vecteur de personnages. Le polymorphisme nous permet une telle opération car les héros et les vilains sont des personnages.

- Afficher ensuite chaque personnage du vecteur de personnages toujours en faisant appel à la méthode `afficher` , encore avec les héros et vilains de couleurs différentes.

- Finalement, créer un `VilainHero` en passant à son constructeur un vilain et héros de votre choix, venant des vecteurs de vilains et héros, en autant que le vilain n'est pas l'ennemi du héro ! Afficher-le à l'écran avec une troisième couleur, par exemple en mauve. L'ajouter aussi au vecteur de personnages pour voir si l'affichage du vecteur fonctionne correctement pour tous nos types de personnages (les héros, vilains et vilains héros sont affichés correctement avec toutes leurs informations, et des couleurs différentes).

- Il ne devrait pas y avoir de ligne non couverte par les tests, sauf l'erreur fatale dans lectureBinaire.cpp (une ligne).

ANNEXE 1 : Structure fichiers binaires heros.bin et vilains.bin

Tous les entiers dans les fichiers sont dans le format lu par la fonction `lireUintTailleVariable` et toutes les `string` sont dans le format lu par la fonction `lireString`.

N'oubliez pas qu'à chaque appel d'une fonction de lecture, vous avancez dans le fichier.

Fichier heros.bin :

Le fichier débute par un entier qui indique le nombre de héros à lire. Par la suite, la même sous-structure se répète pour chaque héro :

1. Une `string` pour le nom du héros
2. Une `string` pour le jeu de parution.
3. Une `string` pour son ennemi.
4. Un entier pour le nombre d'alliés à lire
5. Une `string` pour le nom du premier allié.
6. Une `string` pour le nom du deuxième allié.
7. …
8. Une `string` pour le nom du dernier allié, idem
9. Début de la sous-structure du prochain héros (retour à 1.)

Fichier vilains.bin :

La structure est plus simple, le fichier débute tout d'abord par un entier pour le nombre de vilains à lire, puis chaque sous-structure de vilain consiste en trois strings : nom, jeu de parution et objectif.

Point important ! Si vous utilisez des appels de fonctions de lecture comme paramètres multiples d'une fonction (incluant les constructeurs), en C++ l'ordre d'évaluation des arguments d'une fonction est « non spécifié », donc il pourrait lire dans n'importe quel ordre et ça ne donnera pas le bon résultat.
Exemple : MaClasse(lireA(), lireB()) // On ne sait pas si lireA() sera fait avant ou après lireB() car les deux sont passées au même constructeur, et l'ordre d'évaluation des paramètres d'un même appel n'est pas défini en C++.

Mais l'ordre de construction est bien spécifié (chap.13), et l'évaluation des paramètres dans différentes constructions suit cet ordre.
Exemple de liste d'initialisation : MaClasse() : Base1(lireA()), Base2(lireB()), attribut1(lireC()), attribut2(lireD())
// La lecture se fait dans l'ordre de construction (chap.13) car les lectures sont passées à des constructeurs différents.

ANNEXE 2 : 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 3 : 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)

Mêmes points que le TD3 :

2 : noms des types en UpperCamelCase
3 : noms des variables en lowerCamelCase
5 : noms des fonctions/méthodes en lowerCamelCase
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
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
44,69 : ordonner les parties d'une classe public, protected, private
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