

# INF224

## Travaux Pratiques C++/Objet

**Eric Lecolinet - Télécom ParisTech - Dept. INFRES**

### Liens utiles

- [Page de INF224, Transparents du cours](#)
- **Manuels de référence:**
  - [cppreference.com](#)
  - [cplusplus.com](#)
  - on peut aussi juste chercher la classe ou la fonction sur *Google* !
- **Documentation automatique avec Doxygen :**
  - [Manuel](#)
  - [Tutoriel](#)

**Et aussi :**

- [Toolkit graphique Qt \(en C++\)](#)
- [C++ FAQ](#)
- [StackOverflow](#)
- [Sérialisation avec Cereal](#)
- [Extensions Boost](#)

### Préambule

#### Objectif

Le but de ce TP est de créer l'ébauche du logiciel d'une **set-top box multimédia** permettant de jouer des vidéos, des films, d'afficher des photos, etc.

Ce logiciel sera réalisé par étapes, en se limitant à la déclaration et l'implémentation de quelques classes et fonctionnalités typiques que l'on complétera progressivement. Il est utile de lire le texte de chaque étape en entier avant de la traiter (en particulier les notes ou remarques qui donnent des indications).

#### Rendu

Le rendu aura lieu en fin d'UE. Les deux parties (ce TP) et la partie **Java/Swing** seront rendues en même temps (car elles sont liées). Les instructions pour rendre le TP se trouvent à la fin du TP **Java/Swing**.

Un **README** décrivant brièvement votre travail vous sera demandé ainsi qu'un **Makefile** (cf. plus bas). Pensez à noter au fur et à mesure ce que vous avez fait et les réponses aux questions.

Lorsqu'il y a des modifications importantes du code source (en particulier dans la fonction **main()**) il est utile de conserver la version antérieure en commentaire ou, mieux, entre **#ifdef**

**VERSION\_TRUNC** et **#endif**, comme expliqué dans le TP.

## Environnement de développement (IDE)

Il est indispensable d'utiliser un **outil approprié** pour développer du code C++. Il doit permettre : 1) d'afficher le code correctement (avec coloriage syntaxique), 2) de compiler **dans l'IDE**, 3) (si possible) de déboguer **dans l'IDE**.

Certains environnements n'offrent pas les fonctionnalités 2) et 3), souvent parce qu'ils sont mal configurés. Si c'est un choix personnel, faites comme bon vous semble. Sinon, je vous encourage vivement à changer d'IDE. Les messages des compilateurs C++ sont souvent longs, nombreux, et pas toujours limpides, ce qui nécessite de pouvoir lire le code en même temps que les erreurs. **Un IDE adapté vous fera gagner un temps précieux !**

Sur les machines de l'école (ou la votre si vous l'installez) vous pouvez par exemple utiliser **QtCreator**, qui a l'avantage d'être compatible avec **Makefile**. Il est accessible à l'école depuis le menu *Développement* ou depuis le *Terminal* en tapant la commande **qtcreator &**. Les autres outils ne sont pas nécessairement configurés pour du C++.

**Si vous voulez travailler sur votre machine, pensez à installer l'IDE à l'avance** car cette opération peut prendre pas mal de temps. Pensez aussi à installer le package C++ (e.g. avec Visual Studio) et attention aux versions (e.g. Eclipse pour Java vs. Eclipse pour C++).

## Makefile

Le problème avec les IDEs, c'est qu'ils ne sont pas **portables** (il faut avoir l'IDE sur sa machine pour recompiler). **Nous vous demandons donc de créer également un Makefile**. Ce fichier permettra de tout compiler et même de lancer le programme en tapant **make run** dans le Terminal. **Makefile** est standard sous Unix et Mac et disponible sous Windows.

**Attention un programme sans Makefile sera considéré non rendu** car nous ne pourrions pas recompiler votre programme !

## TP

### 1e Etape: Démarrage

1. Ouvrir une fenêtre **Terminal**
2. Créer un **répertoire** pour les fichiers de ce TP, par exemple : `mkdir inf224`
3. Aller dans ce répertoire : `cd inf224`
4. Copier les fichiers **Makefile** et **main.cpp** dans ce répertoire. **Ne pas faire de coupé collé**, mais utiliser la commande adéquate du navigateur (généralement dans le menu contextuel).
  - Raison : le **Makefile** contient des **tabulations** que le coupé collé transforme parfois en espaces !
  - Attention : si le navigateur rajoute l'**extension .txt** au fichier **Makefile** il faut l'enlever !
5. C'est prêt, vous pouvez lancer l'IDE et importer **main.cpp**.

### Pour créer un projet Makefile compatible avec QtCreator

- Cliquer *"New file or project..."* dans le menu *"File"*
- Cliquer *"Import a project"* puis *"Import an existing project"*

- Choisir un **nom de projet** (ce que vous voulez) et le **répertoire** que vous avez précédemment créé
- Sélectionner ce répertoire en veillant à ce que le fichier **main.cpp** soit sélectionné.
- Valider autant de fois que nécessaire

Ceci va créer un fichier qui a le nom du projet avec l'extension **.creator**. Ce fichier permettra de réouvrir votre projet plus tard en cliquant sur "*Ouvrir un fichier ou projet...*" dans le menu "*Fichier*" (ou, avec certains environnements, juste en double-cliquant sur le fichier **.creator**).

## 2e Etape: Classe de base

Ecrire la **déclaration** (fichier header **.h**) et l'**implémentation** (fichier source **.cpp**) de la **classe de base** de l'arbre d'héritage des classes d'objets multimédia. Cette classe de base contiendra ce qui est commun à tous les objets multimédia. On s'en servira ensuite pour définir des sous-classes spécifiques à chaque type de donnée (par exemple une classe photo, vidéo, film, morceau de musique, etc.)

Pour créer ces deux fichiers, dans **QtCreator** cliquer : *Fichier* puis *Nouveau fichier ou projet...* puis *Classe C++* (le nom du **.h** et du **.cpp** sera forgé à partir de celui de la classe). Noter aussi que, par convention, les noms de vos classes devront commencer par une majuscule et ceux des variables et des fonctions par une minuscule.

Pour simplifier, cette classe de base n'aura que deux variables d'instance:

- Le **nom** de l'objet multimédia qui devra être de type **std::string** (ne **pas** utiliser les antediluvien **char\*** du langage C !). Noter que **std** est le nom du **namespace** de la **bibliothèque standard C++** (dont la classe *string* fait partie).
- Le **nom du fichier** associé à l'objet multimédia. Il s'agit du chemin complet (pathname) permettant d'accéder à ce fichier (une photo jpeg, une vidéo mpeg, etc.) dans le système de fichiers Unix.

Déclarer et implémenter deux **constructeurs** (un sans argument, un avec arguments), le **destructeur**, ainsi que les **accesseurs** ("getters") et **modifieurs** ("setters") pour pouvoir lire ou modifier les variables d'instance.

Déclarer et implémenter une **méthode d'affichage** permettant d'afficher la valeur des variables de l'objet. Cette méthode, qui s'inspirera de l'exemple vu en cours, ne modifiera pas l'objet et affichera les données sur un **std::ostream**, c'est-à-dire un **flux de sortie**. Ceci vous permettra d'utiliser la même fonction pour afficher sur le Terminal, dans un fichier ou dans un buffer de texte, ce qui sera utile plus tard. Concrètement:

- votre fonction doit avoir un paramètre de type **std::ostream &**. N'oubliez pas le **&** on verra à quoi il sert ultérieurement.
- pour afficher sur le Terminal, il suffira que cette fonction ait pour argument **std::cout** ("console out") c'est-à-dire la **sortie standard**.
- votre méthode d'affichage ne doit **pas** créer ou renvoyer une **string** comme le ferait *toString()* en Java (on fera cela plus tard grâce à **std::stringstream**).

**Par ailleurs pensez à :**

- inclure dans votre header les déclarations des objets de la bibliothèque standard dont vous avez besoin, c'est-à-dire au moins les headers **string** et **iostream** (ces headers déclarent les classes **std::string** et **std::iostream**).
- utiliser le mot-clé **const** pour les fonctions là où c'est souhaitable (**toutes** les fonctions qui ne modifient pas les variables d'instances doivent être **const**). Ce mot-clé sert à éviter des erreurs (i.e. à éviter de modifier des variables qui ne doivent pas l'être par inadvertance).

**Pour compiler le fichier et corriger les erreurs**

- Ouvrez le fichier **Makefile** pour modifier:
  - **PROG** : le nom du programme que l'on veut produire
  - **SOURCES** : la liste de vos fichiers **.cpp** séparés par des espaces
  - Ne mettez pas les .h dans SOURCES**
- Si vous utilisez **QtCreator**:
  - vous devrez éventuellement cliquer sur *Liste des Projets* dans la barre du haut puis sélectionner *System Files* pour faire apparaître les fichiers du répertoire de travail (en particulier le fichier Makefile)
  - Cliquer sur le marteau (en bas à gauche) ou **Control-B** lance la compilation. Une barre horizontale indique alors l'état de la compilation : tant qu'elle est verte tout va bien, si elle devient rouge il y a une erreur ! Dans ce cas, cliquer sur la barre pour afficher la liste des erreurs. **Cliquer sur une erreur fait alors apparaître la ligne qui a provoqué l'erreur.**
- Dans tous les cas il y aura une erreur d'**édition de lien** si le fichier **main.cpp** n'existe pas ou si vous ne l'avez pas rajouté aux **SOURCES** du *Makefile* (voir question suivante).

### 3e Etape: Programme de test

Un **programme exécutable** nécessite une fonction **main()**. Cette fonction ne doit pas se trouver dans l'implémentation d'une classe car ceci interdirait sa réutilisation. On va donc l'implémenter dans un autre fichier, ici **main.cpp**.

Pour tester, créer quelques instances de la classe de base (en utilisant **new**) dans **main()** et vérifier que la fonction d'affichage affiche correctement la valeur des attributs dans le Terminal. Noter que votre code doit (et devra par la suite) respecter le **principe d'encapsulation**: **on ne doit jamais accéder aux variables des objets autrement que par des méthodes.**

- **Pour compiler** : faites comme précédemment ou tapez la commande **make** dans le Terminal dans le répertoire du projet.
- **Pour exécuter le programme** : tapez **make run** dans le Terminal, ou tapez le nom du programme précédé de **./** (exemple : **./myprog**)

Pensez :

- A **Indenter** votre code source : avec la plupart des IDEs **Control-I** (ou bien TAB) **indente automatiquement** la ligne courante et **Control-A Control-I** indente tout le texte.
- A rendre votre programme aussi **lisible** que possible :
  - en **séparant** les méthodes par des lignes blanches,
  - en mettant des espaces **autour des =** et **après les ,** et **les ;**
  - en **passant à la ligne** pour que votre code ne fasse pas plus qu'environ 80 caractères de large. Les passages à la lignes n'affectent pas la compilation mais ils rendent le code plus lisible (ce qui réduit les erreurs) !
- A **Commenter**, en particulier dans les headers afin de pouvoir générer automatiquement la documentation grâce à **Doxygen**. On pourra pour se limiter aux fonctionnalités de base de Doxygen.

### 4e Etape: Photos et videos

On va maintenant créer deux **sous-classes** de la classe de base, l'une correspondant à une **photo**, l'autre à une **vidéo**. Ces classes pourraient comprendre de nombreux attributs mais on va faire simple pour ne pas perdre de temps :

- une **vidéo** a une **durée**, c'est-à-dire une valeur numérique intégrale,

- une **photo** peut être caractérisée par un **latitude et une longitude**, c'est-à-dire deux valeurs numériques réelles.

Ces deux classes devront être déclarées dans des fichiers qui leurs sont propres pour obtenir une plus grande modularité et faciliter la réutilisation. Ces classes étant simples, on pourra les **implémenter dans les headers** (auquel cas il ne doit pas y avoir de fichier .cpp).

Déclarer et implémenter les constructeurs, les accesseurs, les modifieurs et la méthode d'affichage (qui doit avoir la même signature que dans la classe parente, **const** compris). N'oubliez pas d'**initialiser les variables qui ont des types de base** dans les constructeurs sinon leur valeur sera **aléatoire** (contrairement aux objets, qui sont initialisés automatiquement grâce à leurs constructeurs).

Enfin, déclarer et implémenter une méthode qui permette de **jouer l'objet multimédia**, c'est-à-dire, suivant le cas, d'afficher la photo ou de jouer la vidéo. Concrètement, cette fonction appellera un autre programme (par exemple, sous *Linux*, "mpv" pour une vidéo ou "imagej" pour une photo) via la fonction standard **system()**, exemple:

```
system("mpv path &");    // path est le chemin complet
```

Pour créer l'argument de **system()** il suffit de concaténer les **strings** avec l'opérateur **+**, puis d'appeler la méthode **data()** de la string résultante pour la convertir en **char \*** (car **system()** prend une **char \*** en argument). N'oubliez pas le **&** afin de lancer l'autre programme en tâche de fond.

Comme pour la fonction d'affichage:

- la fonction pour jouer l'objet ne doit pas modifier l'objet
- elle doit être déclarée dans les classes **Photo** et **Video** mais aussi dans la **classe de base** pour permettre le **polymorphisme**

Contrairement à la fonction d'affichage:

- elle ne peut pas avoir d'implémentation au niveau de la classe de base (car chaque type d'objet nécessite un programme différent pour être joué).

*Comment appelle-t-on ce type de méthode et comment faut-il les déclarer ?*

Modifier le **Makefile** si nécessaire (on rappelle qu'il ne faut pas mettre les .h dans **SOURCES**). Compiler, corriger les éventuelles erreurs et tester le programme.

Si vous avez fait ce qui précède comme demandé, il ne sera plus possible d'instancier des objets de la classe de base. *Pourquoi ?*

## 5e Etape: Traitement uniforme (en utilisant le polymorphisme)

On veut maintenant pouvoir traiter de manière **uniforme** une liste comprenant à la fois des photos et des vidéos sans avoir à se **préoccuper de leur type**.

Pour ce faire créer dans **main.cpp** un **tableau** dont les éléments sont tantôt une photo tantôt une vidéo. Ecrire ensuite une boucle permettant d'afficher les attributs de tous les éléments du tableau ou de les "jouer". Cette boucle doit traiter tous les objets dérivant de la classe de base de la même manière.

- Quelle est la propriété caractéristique de l'**orienté objet** qui permet de faire cela ?
- Qu'est-il spécifiquement nécessaire de faire **dans le cas du C++** ?
- Quel est le **type des éléments du tableau** : le tableau doit-il contenir des **objets** ou des **pointeurs** vers ces objets ? Pourquoi ? Comparer à Java.

Compiler, exécuter, et vérifier que le résultat est correct.

## 6e étape. Films et tableaux

On veut maintenant définir une sous-classe **Film** dérivant de la classe **Video**. La principale différence est que les Films comporteront des **chapitres** permettant d'accéder rapidement à une partie du film. Pour ce faire on va utiliser un **tableau** d'entiers contenant la **durée** de chaque chapitre.

Ecrire la classe **Film**, qui doit avoir :

- le ou les **constructeurs** adéquats.
- un **modifieur** permettant de passer un **tableau de durées** en argument (bien lire les notes ci-dessous)
- un (ou des) **accesseur(s)** retournant le tableau de durées et le nombre de chapitres
- une **méthode d'affichage** affichant la durée des chapitres (la méthode pour jouer l'objet n'a pas besoin d'être redéfinie)

**Notes :**

- Un programmeur averti utiliserait un **std::vector** (que nous verrons plus tard) plutôt qu'un tableau. Ce n'est pas le cas ici car le but de cette question est de montrer certaines difficultés.
- En C/C++, un tableau est référencé par un **pointeur**. Lorsque le nombre d'éléments change, il faut créer un nouveau tableau et détruire le précédent. Lorsqu'il n'y a aucun élément, on ne crée pas de tableau et le pointeur doit valoir **nullptr**.
- En C/C++, quand on passe un tableau en argument à une fonction son **adresse mémoire** est copiée dans le paramètre de la fonction (qui est en fait un pointeur). Ceci a deux conséquences:
  - La fonction ne peut pas **savoir le nombre d'éléments** du tableau à moins de le passer **également en argument** (ou d'utiliser une convention, par exemple les **char \*** du C se terminent toujours par un 0). C'est différent en Java (les tableaux Java connaissent le nombre d'éléments).
  - Les éléments **ne sont pas copiés** (seule l'adresse du tableau est copiée).

Connaissant ce qui précède, la question est de savoir comment le **modifieur** doit **stocker le tableau** dans l'objet Film. Considérez les points suivants :

- Le tableau donné en argument peut être **détruit** ou **réutilisé** dans le futur
- Inversement, l'objet Film doit assurer la **pérénité des ses données**. Si ses données devenaient subitement invalides non seulement le **principe d'encapsulation** serait enfrain (l'objet perdrait le contrôle de ses données) mais le programme aurait de grandes chances de **planter** !

De même, réfléchissez à ce que l'**accesseur** doit retourner. il ne doit en effet pas permettre à l'appelant de modifier le contenu du tableau (ce qui romprait également le **principe d'encapsulation**).

Implementez votre classe et vérifiez que le resultat est correct en modifiant et/ou détruisant le tableau qui est passé en argument puis en appelant la fonction d'affichage de l'objet (NB: il faut répéter ces opérations de sorte que la mémoire soit réutilisée).

**Remarque :** ces questions ne sont pas propres aux tableaux, elles se posent chaque fois qu'un modifieur a en argument un pointeur ou une référence. La question est en fait de savoir **qui possède et contrôle les données**.

## 7e étape. Destruction et copie des objets

Contrairement à Java, C/C++ ne gère pas la **mémoire dynamique automatiquement** : comme il n'y a pas de **ramasse miettes**, ce qui est créé avec **new** occupe de la mémoire

jusqu'à la terminaison du programme, sauf si on le détruit avec **delete**. (Remarque : ce problème ne se pose qu'avec ce qui est créé avec **new**, **delete** ne doit jamais être utilisé dans un autre cas).

Parmi les classes précédemment écrites quelles sont celles qu'il faut modifier afin qu'il n'y ait **pas de fuite mémoire** quand on détruit les objets ? Modifiez le code de manière à l'éviter.

La **copie d'objet** peut également poser problème quand ils ont des variables d'instance qui sont des pointeurs. Quel est le problème et quelles sont les solutions ? Implementez-en une.

## 8e étape. Créer des groupes

On va maintenant créer une nouvelle classe servant à contenir un **groupe** d'objets dérivant de la classe de base. Un groupe peut contenir un ensemble d'objets similaires (e.g. un groupe pour toutes les photos et un autre pour toutes les vidéos) ou pas (e.g. un groupe pour les photos et vidéos de vacances).

Pour ce faire on va utiliser la **classe template** `std::list< >` de la librairie standard qui permet de créer une liste d'objets (dont il faut préciser la classe entre les `< >`). Notez qu'il s'agit d'utiliser une classe template existante, pas d'en créer une nouvelle.

Deux stratégies sont possibles :

- soit créer une classe groupe qui **contient** une liste d'objets (la liste est alors une variable d'instance de la classe)
- soit créer une classe groupe qui **hérite** d'une liste d'objets (la liste est alors héritée de la classe parente)

La première stratégie nécessite de définir des méthodes dans la classe groupe pour gérer la liste. La seconde stratégie évite ce travail car elle permet d'hériter des méthodes de `std::list`. Elle est donc plus rapide à implémenter mais offre moins de contrôle (on ne choisit pas les noms des méthodes comme on veut et on hérite de toutes les méthodes de `std::list` y compris certaines qui sont peut-être inutiles ou pas souhaitables).

Pour aller plus vite, écrivez cette classe en utilisant la seconde stratégie. Définir les méthodes suivantes:

- un **constructeur**
- un **accesseur** qui renvoie le nom du groupe
- une **méthode d'affichage** qui affiche les attributs de tous les objets de la liste

Le groupe ne doit pas détruire les objets quand il est détruit car un objet peut appartenir à plusieurs groupes (on verra ce point à la question suivante). On rappelle aussi que la liste d'objets doit en fait être une liste de **pointeurs** d'objets. Pourquoi ? Comparer à Java.

Pour tester, créez quelques groupes dans `main()` en les peuplant de photos, vidéos ou films et en faisant en sorte que des objets **appartiennent à plusieurs groupes**. Appelez la fonction d'affichage des groupes pour vérifier que tout est OK.

## 9e étape. Gestion automatique de la mémoire

Comme on l'a vu aux étapes 6 et 7, la gestion de la mémoire dynamique (celle allouée avec **new** en C++ et **malloc()** en C) est délicate. On risque en effet, soit de se retrouver avec des **pointeurs pendants** parce que l'objet qu'ils pointaient a été détruit ailleurs (cf. étape 6), soit avec des **fuites mémoires** parce que l'on n'a pas détruit des objets qui ne sont plus pointés nulle part (cf. étape 7).

Les **pointeurs pendants** sont une source majeure de **plantages** ! Les **fuites mémoires** posent surtout problème si les objets sont gros (e.g. une image 1000x1000) et/ou si le

programme s'exécute longtemps (e.g. un serveur Web tournant en permanence). On peut alors rapidement épuiser toute la mémoire disponible (noter cependant que la mémoire allouée de manière standard est toujours récupérée à la terminaison du programme).

Le ramasse miettes de Java et les **smart pointers avec comptage de références** de C++ offrent une solution simple à ce problème : les objets sont alors **automatiquement détruits** quand plus aucun (smart pointer) ne pointe sur eux. Il ne faut donc jamais détruire avec **delete** un objet pointé par un smart pointer !

Le but de cette question est de remplacer les **raw pointers** (les pointeurs de base du C/C++) par des **smart pointers non intrusifs** dans les groupes de la question précédente. Les objets seront alors automatiquement détruits quand ils ne seront plus contenus par aucun groupe. Pour ce faire, utilisez les **shared\_ptr<>** qui sont standard en C++11. Afin de vérifier que les objets sont effectivement détruits, modifiez leurs destructeurs de telle sorte qu'ils affichent un message sur le Terminal avant de "décéder".

### Remarques

- Cette question ne porte pas sur les Films, qu'il n'est pas souhaitable de modifier.
- Pour simplifier l'expression, vous pouvez créer de nouveaux types grâce à **using** ou **typedef** en les déclarant dans le ou les headers appropriés:

```
using TrucPtr = std::shared_ptr<Truc>;
typedef std::shared_ptr<Truc> TrucPtr;
```

Enlevez des objets des groupes et vérifiez qu'ils sont effectivement détruits quand ils n'appartiennent plus à aucun groupe (et s'ils ne sont plus pointés par aucun autre smart pointer : noter que si `p` est un smart pointer `p.reset()` fait en sorte qu'il ne pointe plus sur rien)

## 10e étape. Gestion cohérente des données

On va maintenant créer une classe qui servira à fabriquer et manipuler tous les objets de manière cohérente. Elle contiendra deux variables d'instance:

- une table de tous les objets multimédia
- une table de tous les groupes

Afin de permettre de retrouver efficacement les éléments à partir de leur nom, ces tables ne seront pas des tableaux ni des listes mais des **tables associatives** (cf. la classe template [map](#)). A chaque élément sera associé une **clé** de type **string** qui sera, suivant le cas, le nom de l'objet ou du groupe. Commencez tout d'abord par écrire cette classe, comme d'habitude dans un nouveau fichier. Comme précédemment, on utilisera les smart pointers afin de gérer la mémoire automatiquement.

Déclarer et implémenter des méthodes adéquates pour :

- **Créer** une Photo, une Vidéo, un Film, un Groupe et l'ajouter dans la table adéquate. Noter qu'il faut une méthode pour chaque type, renvoyant l'objet créé, afin de pouvoir le manipuler ultérieurement (c'est-à-dire appeler les fonctions déclarées pour ce type)
- **Rechercher et Afficher** un objet multimédia ou un groupe à partir de son nom, donné en argument. L'objectif sera d'afficher les attributs d'un objet ou le contenu d'un groupe dans le Terminal (s'il existe, sinon afficher un message d'alerte).
- **Jouer** un objet multimédia (à partir de son nom, donné en argument). Même chose que précédemment sauf que l'on appelle la méthode `play()` au lieu d'afficher les attributs.

Pour implémenter ces méthodes vous aurez probablement besoin des méthodes suivantes : **map::operator[]** (pour l'insertion), **map::find()** et **map::erase()**. Faites ensuite quelques essais dans **main.cpp** pour vérifier que ces méthodes fonctionnent correctement.

Les méthodes précédentes permettent d'assurer la **cohérence de la base de données** car quand on crée un objet on l'ajoute à la table adéquate. Par contre, ce ne sera pas le cas si on

crée un objet directement avec **new** (il n'appartiendra à aucune table). Comment peut-on l'interdire, afin que seule la classe servant à manipuler les objets puisse en créer de nouveaux ?

**Question additionnelle** (vous pouvez passer cette question si vous êtes en retard) :

- Ajouter des méthodes pour **Supprimer** un objet multimédia ou un groupe à partir de son nom, donné en argument. L'élément doit être enlevé de la table adéquate et détruit s'il n'appartient plus à aucune table. De plus, lorsqu'on supprime un objet multimédia il faut d'abord l'enlever de tous les groupes dans lesquels il figure.

## 11e étape. Client / serveur

Cette étape vise à **transformer votre programme C++ en un serveur** qui communiquera avec un client qui fera office de télécommande. Dans cette question le client permettra d'envoyer des commandes textuelles. Plus tard, dans le TP suivant, vous réalisez une interface graphique **Java Swing** qui interagira avec le serveur de la même manière. Dans la réalité le serveur tournerait sur la set-top box et le client sur un smartphone ou une tablette.

Récupérez [ces fichiers](#), qui comprennent un client et un serveur ainsi que des utilitaires qui facilitent la gestion des sockets (le client est constitué des fichiers *client.cpp*, *ccsocket.cpp* ; le serveur des fichiers *server.cpp*, *tcpserver.cpp*, *ccsocket.cpp*). Pour les compiler vous pouvez taper : **make -f Makefile-cliserv** dans le Terminal.

Le **serveur** doit être lancé en premier et terminé en dernier.

- Le **client** crée une **Socket** qu'il connecte au serveur via la méthode **connect()**. Celle-ci précise à **quelle machine et à quel port** il faut se connecter. Par défaut le port est **3331** (le même que pour le serveur) et la machine est **127.0.0.1**, ce qui signifie que le client doit tourner sur la même machine que le serveur. La méthode **connect()** renvoie 0 si la connexion réussit et une valeur négative en cas d'erreur.

S'il n'y a pas de firewall bloquant les connexions le **client** peut tourner sur une autre machine à condition de mettre l'adresse de la machine du serveur à la place de 127.0.0.1.

Si la connexion est réalisée, le client lance une boucle infinie (pour quitter, taper ^C ou ^D) qui demande une chaîne de caractères à l'utilisateur puis l'envoie au serveur via la méthode **writeLine()**. Le client **bloque** jusqu'à la réception de la réponse retournée par le serveur qui est lue par la méthode **readLine()**.

- Côté **serveur**, une **lambda** est donnée en argument à **TCPServer()**, le constructeur du serveur. Cette lambda sera appelée chaque fois que le serveur recevra une requête du client. La lambda récupère la requête via son argument **request**, effectue un traitement, puis retourne une réponse vers le client via son argument **response**.

Pour l'instant cette fonction se contente de copier dans **response** "OK:" suivi de la valeur de **request**. **C'est cette fonction qu'il vous faudra adapter** pour qu'elle fasse le traitement voulu.

**Note** : on rappelle que les lambdas peuvent **capturer les variables** de la fonction où elles sont appelées (et le pointeur caché **this** des objets).

Le **client** et le **serveur** utilisent les classes **Socket**, **ServerSocket** et **SocketBuffer** qui permettent de faciliter l'utilisation des sockets :

- **Socket** sert à créer une socket en mode **TCP**
- **ServerSocket** est une socket spéciale qui permet à un serveur de détecter les demandes de connexion des clients

- **SocketBuffer** sert à simplifier la gestion des messages. Dans le cas général, les sockets TCP **ne préservent pas les frontières entre les messages**, ainsi un message peut arriver en plusieurs parties ou au contraire être collé au message précédent. Les méthodes **writeLine()** et **readLine()** de **SocketBuffer** résolvent ce problème grâce à l'addition d'un délimiteur à la fin de chaque message. Par défaut c'est le **caractère '\n'**, **vos messages ne doivent donc pas le contenir.**

Noter que :

- Le client et le serveur utilisent le **protocole connecté TCP/IP** qui permet d'échanger un flux d'octets (stream) entre deux programmes. Ce protocole assure qu'il n'y a pas de paquets perdus et qu'ils arrivent toujours dans l'ordre.
- La connexion est **persistante** : quand on lance le client il se connecte au serveur et reste connecté jusqu'à la **terminaison** du client. Une autre possibilité serait d'établir une nouvelle connexion et de la clore à chaque échange client/serveur (c'est ce fait un navigateur Web par défaut).
- Le dialogue est **synchrone** : le client envoie un message et **bloque** jusqu'à la réception de la réponse. Ça simplifie la programmation mais ce n'est pas idéal car le client va se bloquer si le serveur ne répond pas (ou s'il est long à répondre). C'est particulièrement gênant quand le client est une interface graphique.
- Le serveur utilise des **threads** et peut être connecté à plusieurs clients simultanément. Ceci poserait des problèmes de concurrence si plusieurs clients modifiaient les données du serveur en même temps (mais nous n'aurons qu'un seul client).

### A vous de jouer :

En vous inspirant de **server.cpp**, adaptez votre propre programme pour qu'il joue le rôle du **serveur de la set-top box** et appelle les fonctions adéquates **chaque fois qu'il reçoit une requête du client**. Pour ce faire il faudra définir un **protocole de communication** simple entre votre serveur et le client (**client.cpp** que l'on remplacera plus tard par une interface Java Swing). Il faut au moins pouvoir :

- **rechercher** un objet multimédia ou un groupe et **afficher ses attributs** sur la "télécommande" (c'est-à-dire, pour l'instant, les envoyer au client qui les affichera sur le Terminal)
- **jouer** un objet multimédia (l'objet doit être joué sur la "set-top box", c'est-à-dire le serveur).

### Remarques

- Afin de conserver la version précédente du code dans le main() vous pouvez le mettre entre **#ifdef VERSION\_TRUC** et **#endif** (il y a aussi **#if** et **#else** et il suffit d'écrire **#define VERSION\_TRUC** en haut du fichier pour que ce soit cette version qui soit compilée).
- Pensez à ajouter les fichiers **ccsocket.cpp** et **tcpserver.cpp** à votre projet. Sous Linux modifier la macro **LDLIBS=** de telle sorte que l'on ait **LDLIBS= -lpthread** dans le Makefile.
- Vous aurez besoin de découper la requête en plusieurs mots. Pour ce faire vous pouvez utiliser les méthodes de **std::string** ou un **std::stringstream**. Un **stringstream** est un "buffer de texte" sur lequel on peut appliquer l'**opérateur >>** ou la fonction **getline()** comme avec l'entrée standard **std::cin** ou les fichiers. La fonction **getline()** permet de découper jusqu'à la fin de ligne ou un caractère qui lui est donné en argument. L'**opérateur >>** s'arrête dès qu'il trouve un espace.
- Une solution simple pour renvoyer les attributs d'un objet au client est d'utiliser sa méthode d'affichage en lui passant en argument un (autre) **stringstream** au lieu de **std::cout**. Le **stringstream** peut ensuite être converti en **std::string** via sa méthode **str()**.
- **Attention**, comme dit précédemment, il ne faut pas envoyer les **caractères '\n' ou '\r'** au client ni au serveur (donc enlever les **std::endl**) car ils servent à délimiter les messages

qu'ils échangent. Ces caractères sont également utilisés comme délimiteurs par Java, ce qui sera utile dans la suite du TP.

**Questions additionnelles** (vous pouvez passer ces questions si vous êtes en retard) :

- Vous pouvez rajouter **d'autres commandes**, par exemple chercher tous les objets commençant par ou contenant une séquence de caractères, ou étant d'un certain type, ou encore afficher toute la base ou détruire un objet ou un groupe.
- Un véritable serveur aurait probablement de nombreuses commandes. Afin d'accélérer la recherche des commandes vous pouvez utiliser une **std::map** contenant des **pointeurs de méthodes** ou des **lambdas** (la clé est le nom de la commande, l'attribut est la méthode ou la lambda correspondante).

## 12e étape. Sérialisation / désérialisation

C++ ne propose pas en standard de moyen de **sérialiser** les objets de manière portable. On peut utiliser des extensions pour le faire (par exemple **Cereal**, Boost ou le toolkit **Qt**) ou juste l'implémenter "à la main" dans les cas simples. C'est ce que l'on va faire maintenant pour les tables d'objets multimédia. Notez qu'il est avantageux d'implémenter en même temps les fonctions d'écriture et de lecture, ces deux fonctionnalités dépendant l'une de l'autre.

Ecrivez toutes les méthodes nécessaires en vous inspirant du cours puis testez les dans **main()** en sauvegardant puis en relisant la table d'objets multimédia (on laisse de côté les groupes). On rappelle:

- qu'il faut utiliser **std::ofstream** pour écrire sur un fichier, **std::ifstream** pour lire depuis un fichier et qu'il est nécessaire de vérifier si les fichiers ont pu être ouverts et de les fermer après usage à l'aide de leur méthode **close()**.
- qu'il faut écrire les attributs des objets de telle sorte que l'on puisse ensuite les lire de manière non ambiguë. Faire en particulier attention aux chaînes de caractères, qui peuvent contenir des espaces. La fonction **getline()** résout ce problème.
- que pour lire - et donc créer - un objet multimédia à partir d'un fichier, il faut connaître sa classe. Il faut donc enregistrer le nom de la classe lors de l'écriture. Une solution est de définir pour chaque classe une méthode retournant son nom (sinon voir remarques plus bas).
- qu'il est souhaitable d'implémenter une **fabrique d'objets** (factory) permettant de créer les objets à partir du nom de leur classe.
- enfin, bien sûr il ne faut pas oublier les principes de l'orienté objet : chaque objet est le mieux placé pour savoir comment il doit être lu ou écrit sur un fichier et l'implémentation doit se faire en conséquence

### Questions additionnelles

- Faire en sorte que l'on puisse utiliser les opérateurs **operator<<** et **operator>>** pour sérialiser et désérialiser les objets. Comme ils ne peuvent pas être déclarés comme des méthodes virtuelles des classes, il faut seulement les définir pour la classe de base et appeler une méthode virtuelle (redéfinie pour chaque sous-classe) pour faire le travail.
- Faites en sorte de pouvoir également (dé)sérialiser la table des groupes, en plus de la table des objets. Attention : ne pas dupliquer les objets (qui peuvent appartenir à plusieurs groupes !). Alternativement, vous pouvez utiliser une extension comme **Cereal**.

**Remarque:** C++ fournit un moyen de récupérer les noms des classes via la fonction **typeid()**, mais leur format dépend de l'implémentation et n'est donc pas portable (par exemple **N7Contact8Address2E** pour **Contact::Address** avec **g++**) La solution la plus simple est donc de faire comme conseillé plus haut. Il existe aussi des extensions pour décoder les noms, par exemple avec **g++** :

```
#include <cxxabi.h>

bool demangle(const char* mangledName, std::string& realName) {
    int status = 0;
    char* s = abi::__cxa_demangle(mangledName, NULL, 0, &status);
    realName = (status == 0) ? s : mangledName;
    free(s);
    return status;
}

std::string realname;
demangle(typeid(Photo).name(), realname);
cout << realname << endl;

ptr<Photo> p;
demangle(typeid().name(), realname);
cout << realname << endl;
```

## 13e étape. Traitement des erreurs

La fiabilité des programmes repose sur la qualité du traitement des erreurs en cours d'exécution. Il faut en effet éviter de produire des résultats incohérents ou des plantages résultant de manipulations erronées. Jusqu'à présent nous avons négligé cet aspect dans les questions précédentes.

Exemples:

- si on crée plusieurs groupes ou objets ayant le même nom
- si on supprime un groupe ou un objet qui n'existe pas
- si le nom contient des caractères invalides
- si le tableau de durées d'un Film a une taille nulle ou inférieure à zéro

Il existe essentiellement deux stratégies pour traiter les erreurs. La première consiste à retourner des **codes d'erreurs** (ou un booléen ou un pointeur nul) lorsque l'on appelle une fonction pouvant produire des erreurs. La seconde consiste à générer des **exceptions**.

Dans le premier cas (codes d'erreurs), il est souhaitable que la fonction réalise une action "raisonnable" en cas d'erreur (par exemple ne rien faire plutôt que planter si on demande de supprimer un objet qui n'existe pas !). Cette solution peut poser deux problèmes:

- la propagation des erreurs entre les fonctions n'est pas toujours facile à traiter correctement (l'erreur pouvant se produire dans une fonction appelée dans une fonction elle-même appelée dans une autre fonction, et ainsi de suite).
- les programmeurs, toujours pressés, ont tendance à négliger les codes renvoyés. Ceci peut rendre imprévisible le comportement de programmes complexes et considérablement compliquer le débogage.

La seconde solution (**exceptions**) est plus sûre dans la mesure où les erreurs doivent être obligatoirement traitées via une clause **catch** sous peine de provoquer la terminaison du programme (ou d'interdire la compilation en Java). Cependant, une utilisation trop intensive des exceptions peut compliquer le code et rendre son déroulement difficile à comprendre.

A vous de jouer ! Gerez les principaux cas d'erreurs comme bon vous semble, en utilisant la première ou la seconde stratégie, ou une combinaison des deux suivant la sévérité des erreurs. Mais faites en sorte que votre code soit cohérent par rapport à vos choix et justifiez les dans le rapport et/ou la documentation générée par [Doxygen](#).

**Remarque:** pour créer de nouvelles classes d'exceptions en C++ il est préférable (mais pas obligatoire) de sous-classer une classe existante de la bibliothèque standard. L'exception [runtime\\_error](#) (qui dérive de la classe [exception](#)) est particulièrement appropriée et son constructeur prend en argument un message d'erreur de type **string**. Ce message pourra être récupéré au moment de la capture de l'exception grâce à la méthode **what()** (cf. l'exemple au bas de [cette page](#)).

## Suite

La suite de ce TP (télécommande en Java Swing) est [disponible ici](#).

---

[Eric Lecolinet](#) - <http://www.telecom-paristech.fr/~elc> - Telecom ParisTech