

# Notes de cours C++

Thibaut Marmey

December 19, 2018

## Contents

<b>1</b>	<b>Formation C++ - Décembre 2018</b>	<b>2</b>
1.1	Généralités . . . . .	2
1.2	Première Partie . . . . .	3
1.3	Deuxième Partie . . . . .	4
1.4	Troisième Partie . . . . .	5
1.4.1	Organisation du code . . . . .	5
1.4.2	Inline . . . . .	6
1.4.3	La constance . . . . .	6
1.5	Quatrième Partie . . . . .	6
1.5.1	Données et méthodes statiques . . . . .	6
1.6	Cinquième Partie . . . . .	7
1.6.1	Gestion des excpetions . . . . .	7
1.7	Sixième Partie . . . . .	8
1.7.1	Types de mémoire . . . . .	8
1.7.2	Mémoire dynamique . . . . .	8
1.7.3	Multiplicité optionnelle . . . . .	9
1.8	Septième Partie . . . . .	10
1.8.1	Constructeur par copie . . . . .	10
1.8.2	Polymorphisme & Héritage . . . . .	10
1.8.3	Classe virtuelle . . . . .	11
1.8.4	Classe abstraite . . . . .	11
1.8.5	Constructeurs et destructeurs . . . . .	12
1.8.6	Pointeur de classe . . . . .	12
1.9	Huitième Partie . . . . .	13
1.9.1	Méthode abstraite . . . . .	13
1.9.2	Override C++11 . . . . .	13
1.9.3	Final . . . . .	14
1.9.4	Constructeur hérité . . . . .	14
<b>2</b>	<b>Programmation C++</b>	<b>15</b>
2.1	L'héritage . . . . .	15
2.2	Polymorphisme . . . . .	16
2.3	Design Pattern . . . . .	16
2.4	Template . . . . .	16
2.5	Autre . . . . .	17

# 1 Formation C++ - Décembre 2018

## 1.1 Généralités

- En orienté objet : **Le passé à une influence sur le futur**
- Conception de base : **TDR (test driven requirement)**
  - On part de ce que l'on veut *main()*
  - **couverture fonctionnelle** : on crée juste les méthodes pour compiler le *main()*
  - On implémente les méthodes dans la classe
- **La conception est beaucoup plus importante que le développement**
- *Main()* est le client
  - On ne doit y voir que des actions (via les méthodes des classes)
  - C'est un test, un cas réel
  - Il ne devrait donc pas y avoir de if, for etc
- Les classes et les méthodes créées sont les services
- **La contrainte amène la qualité**
- Ne pas faire de **Smell Code**
  - Duplication
  - Appel méthodes d'autres classes
  - BLOB : classe trop importante par rapport aux autres
  - Surnombre d'arguments
- Avant le code, grosse partie d'analyse pour :
  - comprendre le système
  - concevoir le cahier des charges
  - analyser les relations entre les différents éléments du projets
- Concevoir c'est créer un système EVOLUTIF + ROBUSTE
  - Maintenance (code à modifier)
  - Robustesse (déterministe, si problème on trouve facilement, gestion des exception...)
- **Clean Code** (qualité logicielle) :
  - méthode courte
  - nom clair
  - économie des messages (méthodes)

- Si méthode trop longue on extrait des lignes de codes pour créer une nouvelle fonction et ensuite placer ces nouvelles méthodes en "private" car ces méthodes ne sont pas utilisées par l'utilisateur !
- En C++ on est proche des objets réels, leur conception se fait en les conceptualisant
- Encapsulation : dans un même objet il y a des données et des fonctions qui peuvent ne pas être accessibles depuis l'extérieur.
- La documentation doit être automatisée par un logiciel : les mises à jour se font au fur et à mesure de l'avancement du code (la doc ne devrait pas être une activité humaine)
- Représenter les choses dont on a besoin : **UML**
  - Toutes les relations entre les classes "All" (pour celui qui développe le système)
  - Seulement avec les choses "Public" (pour celui qui utilise)
  - Il y a toujours un seul "system", c'est celui qui englobe tout le reste
- Documentation CodeOrganization : permet de connaître, visualiser les dépendances entre les différents fichiers
- Raccourci Visual Studio

```
// ctrl + k -> ctrl + c (comment)
// ctrl + k -> ctrl + u (uncomment)
```

## 1.2 Première Partie

- **C++11** - Initialisation uniforme, donne la valeur par défaut avec les accolades, peut importer le type donné

```
int i {};  
float f{};  
char c{};  
class A a{};  
  
// Dans boucle for  
for(int index{};...)
```

- La portée des variables est limitée dans le bloc
- Dans la fonction *main()*, *argc* est toujours supérieur ou égal à 1 car le premier argument est nécessairement le nom de l'application avec son chemin global.

```
int main(int argc, char** argv){  
    // argc = arg count >= 1  
    // argv = arg values  
}
```

- Enumeration fortement typée

```
enum class A { }
// Avant on ne pouvait pas avoir deux enum COLOR{VERT, blabla} et
// ETAT{VERT, blabla}. Il y avait un conflit entre les deux variable
// "VERT".
```

- La boucle FOR EACH

```
for (int n : tab) { } // Par recopie
for (int& n : tab) { } // Par reference (on modifie directement les
// elements)
Dice tab[10] // Tableau de 10 objets Dice
for (Dice& d : tab) // Parcours de tab par reference
```

- La boucle DO-WHILE (passe au moins une fois dans la boucle, contrairement au *while* simple)

```
do{ blabla bla } while(condition)
```

Pour un tableau de taille fixe utiliser *array*

```
array<type, size> name{values...}
```

- Listes d'initialisation génériques : fonctions à paramètres variables

```
#include <initializer_list>

int somme (initializer_list<type> liste) {
// bla bla
}
// appel de la fonction : somme( {a,b,c,d} );
```

## 1.3 Deuxième Partie

- C++11 - utiliser USING pour créer un nouveau nom de type de variable

```
using Vitesse = int;
```

- Utilisation de *this* si conflit entre méthode de la fonction et attribut de la classe

```
this->altitude = altitude;
// this->altitude : pointeur sur l'objet de la classe
// altitude : argument de la fonction
```

- Les constructeurs (pas de type de fonction car ce n'est pas nous qui appelons le constructeur)

- Toujours initialiser les membres de la classe dans la **ZIM : zone d'initialisation des membres**.
- Self-delegation : la classe s'envoie un message à elle toute seule (permet d'éviter le smell code)  
Exemple : dans la classe Dice, on initialise la faceValue par la méthode roll().
- RAND & SRAND pour créer des séquence aléatoire. Il faut utiliser *srand* pour réarmer le seed de rand

```
#include <ctime>
srand(time(nullptr)); // Ne pas mettre dans le main
```

- ".setup" : pour modifier l'objet qui a été créé. C'est une méthode très utilisée (ex interfaces graphiques ou jeu démineur).  
Le jeu (objet) est d'abord créé puis on modifie sa configuration (beginner, expert,...) et on actualise le jeu via la méthode de l'objet jeu ".setup".

## 1.4 Troisième Partie

### 1.4.1 Organisation du code

- Importation des fichiers systèmes (bibliothèques standard C++) avec les chevrons
- Importation des fichiers locaux (les .h) avec les guillemets
- Création de fichiers d'en tête : les fichiers ".h"
  - On stipule que le fichier doit être inclu qu'une seule fois

```
#pragma once
```

- Le pré-processeur fait la concaténation des fichiers avec le *#include*.  
Pas de ";" pour le pré-processeur.  
*#define* est utilisé en C mais il faut arrêter en C++.
- Séparer le contrat ".h" de l'implémentation ".cpp".
  - Le contrat : décrit les fonctions mais pas de code (dit ce dont tu as besoin en entrée)
  - l'implémentation : décrit le code des méthodes
- Opérateur de résolution de porté "::"
- Bonne pratique : créer le fichier "util.h" pour inclure des bibliothèques toujours utilisées, des *using std::*, ou le petit matériel comme les *using*. L'inclure dans le grain de classe le plus faible.

```
#include <string>
using std::string
```

- Ne pas inclure de *using namespace* dans un contrat (.h) car on donne l'accès à beaucoup trop de choses et donc risque de collision, etc...  
On peut donc inclure les *using namespace* dans les .cpp

### 1.4.2 Inline

- Fonction/Méthode **inline**, c'est une fonction implémentée directement dans le contrat ".h". Ce sont des méthodes accesseur de data, celle qui retourne directement une data.

```
// Dans le .h
int getData() {return data}

// On ne fait pas de methode inline si la valeur retournee est une
// fonction mathematique complexe
```

### 1.4.3 La constance

- La constance : si on sait que la fonction ne doit pas avoir d'effet de bord on utilise *const*

```
type nomMethode(...) const {bla bla}
```

La constance est différente du const pour les variables.

## 1.5 Quatrième Partie

### 1.5.1 Données et méthodes statiques

- Les données et méthodes statiques : ce sont des éléments partagés par toutes les instances de la classe.  
Ex avec la classe Voiture :

- attribut de classe : la vitesse max sur les routes en France

```
static inline type nomData // C++17
```

- attribut d'objet : la couleur d'une voiture

- *Static* n'a rien à voir avec "ne bouge pas". Ça veut dire que ça appartient à la classe.
- Une méthode statique peut seulement accéder aux attributs statiques

```
static nomMethode() {return (data)} // Avec data un membre static inline
```

- Utilisation dans le *main()* (c'est un message envoyé à la classe et non à une instance de classe)

```
nomClasse::nomMethode();
```

- On ne peut donc pas utilisé de *this* dans un méthode statique
- Exemple d'utilisation : connaitre le nombre d'instances existants d'une classe
  - utiliser attribut de classe
  - Incrémenter dans le constructeur
  - décrémenter dans le destructeur
  - Récupérer attribut de classe via une méthode de classe
- **C++11** - On ne peut pas initialiser la variable statique directement dans le .h. Il faut passer par le .cpp

```
type nomClasse::varStatique{value}
```

- Dans la méthode statiques utiliser les "::" pour modifier l'attribut statique

## 1.6 Cinquième Partie

### 1.6.1 Gestion des excpetions

- Phase de développement

1-Maquette	2-Prototype	3-Livvable
pas de code	code nominal	rajouter les cas d'erreurs

- **Programmation défensive** : faute de programmation est différente de faute de l'utilisateur (situation métier qui doit être pris en compte)
- Faute de programmation : pour corriger un bug de conception on utilise des ASSERT.
  - On peut mettre des assert de partout, ensuite régler les problèmes.q
  - Utiliser *NDEBUG* pour ne plus exécuter les assertion

```
#include <cassert>
// #define NDEBUG
assert(condition)
```

- Faute de l'utilisateur : pour gérer les exception (**pur C++**)

```
// Dans util.h
#include <exception>
using std::exception

// Dans methode
throw exception
```

```
// Dans main()
try{}
catch{}
```

- Si l'exception est trouvée le programme stop. Le runtime tue le programme si on n'intercepte pas le problème.
- Il faut donc utiliser un gestionnaire d'exception (avec les blocs try et catch)
- On ne peut donc pas ignorer une exception: On doit la *catch*

- Exemple de prise en compte d'une exception :

```
doThis(){
    throw exception{"erreur"}
}

// Dans main()
//...
try{ bla bla
    doThis; }

catch(const exception& e){
    cerr << e.what() << endl; // cerr a la place de cout pour les
    messages d'erreur
}
```

## 1.7 Sixième Partie

### 1.7.1 Types de mémoire

- 3 durées de vie
  - Un objet global : c'est ok pour la durée de vie du programme (destruction à la fin)
  - Un objet dans un bloc : mémoire automatique de bloc
  - Un objet anonyme : est détruit après son utilisation

```
//nomClass{}
A{}.doIt() // Constructeur
// Destructeur apres utilisation
```

### 1.7.2 Mémoire dynamique

- On parle de HEAP = TAS quand on est en dynamique
- On choisit la vie et la mort de l'objet avec *new* et *delete*.

```
A* a{new A{}};
delete a;
```



- On utilise `delete[]` pour les tableaux

```
A* tab = new A[3];
delete[] tab;
// Si seulement delete tab on delete la premiere case, donc un seul
// appel du destructeur
```

- Pour une gestion rigoureuse :  $1new = 1delete$
- Quand on fait du temps réel : c'est interdit de gérer la mémoire dynamique par ce que ce n'est pas déterministe.

### 1.7.3 Multiplicité optionnelle

- Le pionneur `nullptr` est pris en compte dans le C++, on peut donc delete un pointeur `nullptr`. (enfait ça ne fera rien mais il n'y a pas d'erreur)
- Exemple avec la classe Voiture : Y a t-il une climatisation dans la voiture ? Si oui on crée la var `pclim` si non on ne l'a crée pas.
- On utilise donc un pointeur pour `pclim` pour pouvoir la créer ou la détruire quand on veut

```
class Voiture {
    Climatisation* pclim{};
    ...
}
```

- On peut ainsi créer la `pclim` dans le constructeur avec un `new` dans la ZIM (ne pas oublier de delete la `pclim` dans le destructeur s'il y a eu un `new`)
- Autre exemple : avec le projet Game. On veut que l'utilisateur puisse choisir lui-même le nombre de dés.
  - On va construire le gobelet dans avec la méthode "setup". (on ne peut pas créer un gobelet tant que je ne connais pas le nombre de dés qu'il y a dedans)
  - Pour faire cela on va travailler avec un pointeur de Cup (gobelet).
- Utilisation du HEAP = Je veux maîtriser la date de naissance de l'objet.
- Les changements :
  - Dans "setup" de Game

```
cup = new Cup{nbDices}; // Il faut donc un delete quelque part en
// plus
```

- On détruit Cup dans le destructeur

```
delete cup;
```

- Créer un nouveau membre pointeur de Dice pour créer tableau de dés

```
// Dans la classe Cup
Dice* dices{nullptr};
```

- Dans le constructeur on va vraiment créer le tableau de dés

```
dices = new Dice[nbDices] // 1 delete[] dices dans le destructeur
```

- Le jeu (ici Game) ne connaît pas "Dice" et ne sait pas que c'est un objet. Il sait seulement ce qu'est "Cup".  
Comme le conducteur d'une voiture : il sait ce qu'est une voiture mais n'a pas besoin de savoir comment marche le moteur ou le vilbrequin pour pouvoir l'utiliser.

## 1.8 Septième Partie

### 1.8.1 Constructeur par copie

- Problème : Si on initialise un tableau, un objet par copie il vont partager les mêmes données.

```
Cup cup1;
Cup cup2{cup1};
```

Donc quand on fait un delete du premier élément ça marche bien, les données sont bien delete. Or il y a deux objets. Donc pour le deuxième delete il y a une erreur puisque les données n'existent déjà plus.

```
delete cup1; //OK
delete cup2; //ERREUR
```

**Double Délétion = BUG**

- Bloquer l'utilisation du constructeur par copie

```
nomClasse(const nomClass& src) = delete; // Contrat du constructeur par
    copie
// Grace au delete, pas d'utilisation du clonage
// Conseil : faire ca pour toutes les classes qui ont un/des pointeur.s
```

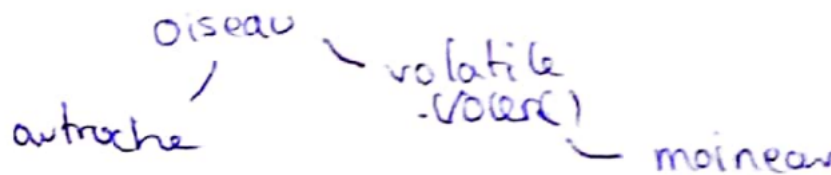
### 1.8.2 Polymorphisme & Héritage

- Hiérarchie de spécialisation ou de généralisation (même chose dans le sens opposé)

Bibliothèque	Document	Video/Livre
– > – >	spécialisation	– > – >
< – < –	généralisation	< – < –

- Les classes héritent des "dettes" des classes mères : les choses qu'il faut respecter pour que ça marche
  - les arguments des constructeurs

- appeler le constructeur de la classe mère dans la classe fille
- rajouter dans la ZIM les attributs qu'il faut initialiser ensuite
- On acquiert le "patrimoine" : pas besoin de self délégation (on hérite des méthodes de la classe mère)
- Si l'on fait une classe fille c'est pour acquérir 100% du patrimoine de la classe mère.  
Ex : Classe Oiseau avec sous classe Autruche et Moineau. On ne peut pas mettre de méthode ".voler()" dans oiseau car une Autruche est un oiseau qui ne peut pas voler.  
On va donc créer une autre classe descendante qui sache voler. On obtient le schéma suivant :



### 1.8.3 Classe virtuelle

- C'est lorsque l'on veut déléguer le traitement
- Je veux appeler la méthode de l'objet qui est en mémoire
- Je veux que le traitement appelé soit fait sur la classe concrète/tangible

### 1.8.4 Classe abstraite

- **Classe abstraite** : c'est une classe que l'on ne peut pas instancier. C'est donc pour les objets non tangibles.
- On peut créer une instance d'une classe abstraite seulement via un pointeur
- Un pointeur :
  - c'est une adresse (c'est pas nouveau)
  - c'est aussi maintenant **un accès à une couverture fonctionnelle** (nouveau)
- Ex : comme la classe mamifères  
Les objets tangibles seraient chien, chat, homme, etc...
- Si on crée un pointeur dans le *main()*, on utilise la syntaxe copie

```

// Dans le main()
Figure* c = new Cercle{0,0,100};
// Instanciation d'un objet de la classe fille Cercle via un pointeur de
// la classe mere Figure

c->methode1();
c->methode2();
// Utilisation de -> pour utiliser les methodes avec un pointeur

```

- On met le constructeur en `protected` : le `main()` ne peut pas instancier d'objet de la classe abstraite

### 1.8.5 Constructeurs et destructeurs

- Schéma de pile : on parle de chaînage
- Par chaînage, les destructeurs des sous-classes sont appelés en premier suivi du destructeur de la classe mère
- Pour une classe concrète/tangible, le destructeur est public (on peut l'utiliser dans `main()`).
- Le destructeur de la classe `Figure` est public sinon il faudrait avoir un destructeur spécifique pour chaque classe fille : **destructeur virtuel**
- Pourquoi le destructeur de la super classe doit être virtuel ?
  - Dans le cas d'une création d'un pointeur de la classe mère pour avoir une instance de la classe fille

```
Figure* r = new Rectangle{...};
```

Si on fait un `delete r` qui est appelé ?

- SANS VIRTUAL : c'est seulement le destructeur de `Figure` car `r` est un pointeur de `Figure`  
DÉLÉTION INCOMPLÈTE
- AVEC VIRTUAL : on va déléguer cette destruction au destructeur de la classe tangible (sous-classe)  
Les destructeurs sont chaînés donc pas besoin d'appeler de destructeur dans le destructeur  
Au final : 1) le destructeur de `Rectangle` est appelé, 2) Par chaînage le destructeur de `Figure` est appelé  
DÉLÉTION COMPLÈTE

### 1.8.6 Pointeur de classe

- Pourquoi utiliser un pointeur de `Figure` pour instancier un `Rectangle` ou un `Cercle`
  - aspect **Evolutif** : si on rajoute une sous-classe on pourra la manipuler de la même façon
    - \* Exemple avec les classes `Zoo`, `Animal`, `Gazelle`, `Lion`
    - \* La méthode `"enfermer()"` de `Zoo` permet d'enfermer **tous** les animaux donc on utilise la classe `Animal` pour ne pas spécifier quel animal à enfermer

```
zoo::enfermer(Animal& animal);
```

- \* On veut avoir une liste d'animaux présents dans le zoo + cette liste peut changer

```
// attribut dans Zoo.h  
vector<Animal*> animaux;
```

- \* On peut utiliser un FOR EACH dans les méthodes pour traiter tout le monde

- **Maintenance**

- Permet de ne pas spécifier quelle type de donnée on crée
- Permet d'avoir une fonction unique pour toutes les sous classes en utilisant un pointeur de la classe mère

```
void destroy(Figure* f) // On peut appeler un Rectangle ou un Cercle
```

- **Polymorphisme** : on s'adresse à des gens différents de la même façon  
Ex: les feux tricolores veulent dire la même chose pour les automobilistes, les cyclistes, les routiers, etc...

## 1.9 Huitième Partie

### 1.9.1 Méthode abstraite

- Une méthode abstraite est une méthode virtuelle (*virtual*) pure (*=0*)

```
virtual type nomMethode() = 0; // methode abstraite
```

- Une méthode abstraite doit obligatoirement être créée et implémentée dans les classes filles.
- MAIS une fonction abstraite peut être implémentée dans la classe mère : elle sera générale pour toutes les sous-classes
- On fera le chaînage à la main pour appelé la méthode de la classe mère

### 1.9.2 Override C++11

- Aspect **Robustesse**
- C'est une sorte de détrompeur
- Override stipule que l'on est entrain de redéfinir une méthode de la classe mère
- Le compilateur va nous avertir d'une erreur si l'on fait override sur une méthode qui n'existe pas dans la classe mère (erreur de nom, type, arguments,...)
- Le contrat de la méthode doit être spécifiquement le même

### 1.9.3 Final

- Aspect **Robustesse**
- Permet de spécifier que l'on ne veut pas implémenter une nouvelle fois une classe, une méthode etc
- *Final* avec les classes : stipule que la classe dit Final ne peut pas avoir de classe fille

```
class A final {...};  
class B : public A {...} // Erreur car A final
```

- *Final* avec une méthode d'une classe : stipule qu'on ne peut pas implémenter la même méthode (contrat identique) dans une classe fille

```
class A {  
public:  
    virtual void f() final;  
}  
class B : public A{  
public:  
    void f(); // Erreur car void f() est final  
}
```

### 1.9.4 Constructeur hérité

- Pour générer tous les constructeurs de la classe mère dans la classe fille MAIS il faut qu'il y ait la même signature (pas de données supplémentaires dans les classes filles)
- utiliser le mot clefs *using* et générer les constructeurs

```
// Dans la classe fille (avec meme signature)  
using classeMere::classeMere; //genere tous les constructeurs
```

## 2 Programmation C++

### 2.1 L'héritage

- Cours openclassroom : héritage
- Déclaration d'une classe héréditaire :

```
class Classe_fille : public|protected|private Classe_mere1
[, public|protected|private Classe_mere2 [...]]
{
    /* Definition de la classe fille. */
};
```

- les données publiques d'une classe mère deviennent soit publiques, soit protégées, soit privées selon que la classe fille hérite en public, protégé ou en privé.
- La classe fille possède les attributs et les méthodes de la classe mère. Elle possède en plus de cela ses propres attributs et méthodes
- Possibilité de surcharger les méthodes de la classe mère dans la classe fille
- Surcharge du constructeur : on peut appeler le constructeur de la classe mère dans le constructeur de la classe fille.  
Pour déclarer un constructeur dans le .h : il doit avoir le même nom que la classe, et il ne doit rien renvoyer. *maClasse()*  
On écrira aussi le constructeur dans le .cpp de la manière suivante :  
*maClasse::maClasse() : blaba, bla, bla { }*
- Masquage de fonctions de la classe mère. On peut substituer le nom d'une fonction présente dans la classe mère et utiliser sous le même nom une méthode spécifique dans une classe fille. On peut toujours appeler la méthode de la classe mère dans la méthode de la classe fille en spécifiant l'appel à la classe mère grâce au double deux points "nomClasseMere::nomMéthode()".
- Dérivation de type : on peut substituer un objet de la classe fille à un pointeur ou une référence vers un objet de la classe mère. On peut affecter un élément enfant à un élément parent  
Il est possible d'écrire *monPersonnage = monGuerrier* car un guerrier est un personnage. L'inverse n'est pas possible ~~*monGuerrier = monPersonnage*~~
- La dérivation de type est très pratique dans l'appel d'un élément dans une fonction par exemple. Si l'argument à mettre est de la classe *Personnage* il est alors possible de mettre tous les autres classes filles de *Personnage*  
*void coupDePoing(Personnage & cible) const; : cible* peut être de type *Personnage* ou *Guerrier*
- Le type *protected* permet aux attributs d'être accessible par les classes filles et inaccessible de l'extérieur

## 2.2 Polymorphisme

- le polymorphisme est un mécanisme dynamique permettant, par voie d'héritage, de spécialiser dans des classes dérivées les comportements annoncés ou implémentés dans des classes de base, indirectes ou non.
- **Résolution statique des liens.** La fonction reçoit un type de data, c'est donc toujours les méthodes de ce type qui sera utilisée. C'est donc le type de la variable qui détermine quelle fonction membre appeler et non sa vraie nature.

*Ex pratique :*

```
void presenter(Vehicule v) //Présente le véhicule passé en argument
{ v.affiche(); }
```

La fonction reçoit un véhicule (classe mère) c'est donc les méthodes de véhicule qui sont appelées même si une surcharge de méthode est présente dans la classe fille.

- **Réolution dynamique des liens.** Lors de l'exécution le programme utilise la bonne version des méthodes car il sait si l'objet est de type mère ou type fille.
  - utiliser un pointeur ou une référence
  - utiliser des méthodes virtuelles
- Il faut placer un pointeur ou une référence comme argument dans la fonction *void presenter(Vehicule const& v)*
- Rajouter *virtual* devant la méthode dans la classe mère (seulement dans le .h) et c'est optionnel dans les classes filles.

## 2.3 Design Pattern

- Ce sont des modèles théoriques adaptables qui résolvent un problème précis.
- **Un prototype :** un prototype est une classe dont le but est d'être clonée.
- **Le singleton :** permet de s'assurer qu'il n'existe qu'une unique instance d'une classe donnée.  
Est une variable globale.
- **La fabrique :** classe dont le rôle est de créer d'autres objets.
- **Les décorateurs :** sont l'ensemble des classes permettant d'étendre dynamiquement le rôle d'une classe de base.

## 2.4 Template

- Les templates sont des fonctions spéciales qui peuvent être utilisées avec des types génériques. Cela nous permet de créer une fonction template dont l'utilisation n'est pas réstainte à un seul type de données, sans répéter le code entier pour chaque type.

```
template <class myType>
myType GetMax (myType a, myType b) {
    return (a>b?a:b); }
```



- Pour utiliser les fonctions templates on utilise le schéma suivant :

```
//function_name <type> (parameters);
int x,y;
GetMax <int> (x,y);
```

- Utilisation des templates avec les classes :

```
template <class T>
class mypair {
    T a, b;
public:
    mypair (T first, T second)
        {a=first; b=second;}
    T getmax ();
};

template <class T> //T for template parameter
T mypair<T>::getmax () //1st T for the type return by the function
//2nd T for requirement to specify the function's template parameter
{
    T retval;
    retval = a>b? a : b;
    return retval;
}
```

## 2.5 Autre

- Utilisation de `=` dans un *if*.

```
int a = 1
int b = 2;
if ((a=b)) {...} //is true
blabla...
int b = 0;
if ((a=b)) {...} //is false
```

- Utilisation de *auto* (C++11) : cet outil permet de spécifier automatiquement le type de la variable en jeu.  
*ex : auto x = 1; auto y = 3.1; auto z = 'a';*  
*output : x est int, y est double, z est char*
- *Lambda function* : *ex : auto allowed = [ℰ](int x, const std::vector<int>ℰvect){...}*
  - a. [=] capture all variables within scope by value
  - b. [&] capture all variables within scope by reference
  - c. [& var] capture var by reference
  - d. [&, var] specify that the default way of capturing is by reference and we want to capture var

- e. [=, & var] capture the variables in scope by value by default, but capture var using reference instead.