Chapitre 3: Héritage en C++

Martin Quinson

Épisode précédent 1
A imprimer: code fourni . . . 1

I) Héritage 2
I.1) Principe . . . 2
I.2) Vocabulaire . . . 2
I.3) Modalité d'accès . . . 2
I.4) Ordre d'appel des ctor, dtor . . . 3
I.5) Heritage multiple . . . 3

II) Polymorphisme d'objets 3
II.1) Principe de substitution de Liskov . . . 3
II.2) Redéfinition de méthode . . . 3
II.3) Types objets en C++ . . . 5

III) Design OOP 6
III.1) Formalisme UML . . . 6
III.2) Programmer proprement . . . 6

IV) Exception 7

Épisodes précédents

- L'encapsulation C++ en pratique
  - classe, constructeur et destructeur. Initialisation de champs
  - champs statiques, champs constants. Méthode statique, constante.
  - Visibilité: private/public
  - Surcharge d'opérateurs
- Gestion de la mémoire en C++
  - Objets statiques, dynamiques, automatiques et temporaires
  - Opérateur de copie
  - Initialisation et copie implicite
  - Tableaux d'objets
- Voir aussi: `https://programmation-orientee-objet.pages.ensimag.fr/poo/resources/fiches/05-Polymorphisme_liaisonDynamique/`
- TODO: ajouter quelques mots sur la représentation mémoire de l'héritage (p452 du reader de Stanford). Ca explique bien pourquoi on peut pas spécialiser une affectation sans pointeurs.

I) Héritage

I.1) Principe

- On a dit la semaine dernière que l'encapsulation était l'un des principes fondamentaux de l'OOP, avec l'héritage et le polymorphisme. Continuons avec l'héritage.
- Objectif: factoriser du code pour réduire sa taille / réutiliser du code existant pour en écrire moins
  - On veut avoir suffisament peu de code pour que tout tienne dans la tête à la fois.
  - Le grand démon est le copie-colle de code: on va vite se perdre dans des kilomètres de code presque pareil, à quelques nuances près. Et il faudra tout maintenir en double: appliquer chaque amélioration/correction à toutes les copies.
- Supposons qu'on veuille modéliser deux especes animales: le pangolin, et le pangolin à longue queue.
  - Un pangolin a un nom, un nombre d'écaille et sait crier.
  - La seule différence d'un pangolin à longue queue est qu'en plus il a une longueur de queue
  - Le code des deux classes s'annonce répétitif si on n'y prend garde.
- Observons le code fourni à la place
  - La classe `PangolinALongueQueue` est une extension / spécialisation de `Pangolin`
  - Ca se voit `PangolinALongueQueue.hpp:3`: le nom de la classe dérivée est écrit après `:`
  - On précise `public` pour signaler que le contenu de `PangolinALongueQueue` voit le contenu de `Pangolin` comme si c'était défini chez lui. On peut aussi hériter sans montrer son implémentation à ses classes filles.
  - En résumé, un héritage permet de faire comme un copie/colle, mais en mieux

I.2) Vocabulaire

- **Classe mère** ou super-classe. Ici `Pangolin`
- **Classe fille** ou sous-classe. Ici `PangolinALongueQueue`
- On parle de **hiérarchie de classes**
- On dit que la classe fille **spécialise**, ou **dérive** de la classe mère.
- On dit que la classe mère **généralise** la classe fille. `Pangolin` regroupe les caractéristiques communes à tous les sous-types de pangolins.
- On dit aussi que `PangolinALongueQueue` **est-un** `Pangolin`

I.3) Modalité d'accès

- Les champs privés de la classe mère sont inaccessible à la classe fille. L'encapsulation est respectée, même dans la hiérarchie
- Si `class PangolinALongueQueue : public Pangolin`, les champs publiques de Pangolin sont publiques dans la fille
- Si `class PangolinALongueQueue : private Pangolin`, les champs publiques de la mère deviendraient privés à la fille. La fille les voit mais pas les utilisateurs de la classe fille.
- Nouveau type d'accès: protected en plus de private et public. C'est visible depuis les sous-classes.

I.4) Ordre d'appel des ctor, dtor

- Si aucun ctor n'est donné, le compilateur en fait un par défaut, qui donne les valeurs par défaut aux champs. Si les champs sont des objets, il faut que les classes correspondantes aient des constructeurs par défaut elles-même.
- Le constructeur de la fille doit choisir un ctor de la mère dans son initialisation (sauf si y'a des ctor sans paramètre)
- Le constructeur de l'ancêtre le plus haut appelé en premier, puis dans l'ordre. On construit un batiment en partant des fondations de base, puis on monte en abstraction.
- Les destructeurs sont invoqués dans l'ordre contraire

I.5) Heritage multiple

- On peut donner plus d'un ancêtre quand on défini une classe, mais c'est souvent une mauvaise idée
- Les champs de chaque ancêtre sont disponibles dans la classe fille, MAIS
- Si les deux ancêtres définissent tous deux une methode d'un nom donné, on ne sait pas laquelle sera invoquée et il faut spécifier dans la liste d'initialisation du constructeur
- C'est particulièrement bête si on a un schéma d'héritage en diamant comme dans le code fourni: l'ancêtre commun est dans les deux branches, donc le constructeur de l'ancetre est appelé 2 fois
  - On peut utiliser un héritage virtuel comme suit pour éviter que le ctor soit appellé 2 fois.
  - Mais c'est souvent une meilleure idée de juste pas faire de diamants, puisque d'autres pbs arriveront dès qu'on oublie de spécifier laquelle des classes mères utiliser dans les fctions virtuelles
- Java interdit l'héritage multipe (sauf pour les interfaces qui sont des classes complétement abstraites, puisqu'elles n'ont pas de code)
  - Scala et Rust l'autorisent, mais que pour les traits avec un ordre de résolution explicite des méthodes

II) Polymorphisme d'objets

II.1) Principe de substitution de Liskov

- On peut affecter un objet dans une variable moins spécialisée: `A* a = new B(3)`.
  - L'utilisateur veut un outil; un marteau saura faire tout ce qu'on peut demander à un outil.
  - Attention, `A a = B();` (sans étoile ni new) est un piège du C++, on y revient la semaine prochaine.
- En revanche l'inverse ne peut pas marcher puisque l'objet général affecté dans une variable plus spécialisé ne saura pas faire les choses de spécialité.
  `B* b = new A(3)` interdit: tous les outils ne font pas de bons marteaux.
- La même règle s'applique sur le passage de paramètre
- Voir si deux objets sont de même type: `typeid(a) == typeid(b)`
- Voir si des objets sont compatibles : C++ introduit de nouveaux transtypages, sur lesquels on revient.

II.2) Redéfinition de méthode

II.2.a) Principe

- On veut parfois modifier une méthode dans la classe fille
- En java, il suffit de définir une méthode de même signature, éventuellement en ajoutant `@Override`

- En C++, il faut déjà que la mère déclare que cette méthode est redefinissable, avec `virtual`
  - La fille peut ensuite redéfinir, éventuellement en ajoutant `override` (bonne idée de laisser le compilateur vérifier qu'on a pas fait de typo dans le nom ou le type des paramètres)
  - Il est interdit de redéfinir une méthode qui n'est pas marquée `virtual` car le compilateur n'installe pas le mécanisme qui permet la redéfinition (la `vtable` cf juste ci-dessous)
  - Au besoin l'implémentation de la fille peut invoquer l'implémentation de la mère avec `mere::methode()` pour l'étendre au lieu de la remplacer
  - Une méthode redéfinie (ou dérivée) peut elle-même être redéfinie dans une classe fille (sauf si elle est marquée `final`, qui casse la possibilité de dériver)

II.2.b) Exemple du code des Marsouins

II.2.c) Override, overload, overwrite

- Quand deux méthodes ont le même nom, plusieurs cas sont possibles en C++.
- Si c'est dans les classes mère/fille, que la méthode ancêtre est marquée `virtual` et que la signature est inchangée, c'est `override`
- Si la signature est changée, c'est de la surcharge (overload), c'est à dire que deux méthodes co-existent avec le même nom et des paramètres différents. Cela n'implique pas d'héritage, et c'est ça qu'on fait depuis longtemps pour les opérateurs.
- Mais il y a un twist en C++ : l'overload est forcément au sein de la même méthode, jamais intergénérationel.
  - Si l'override échoue (soit pas de `virtual`, soit pas exactement les mêmes paramètres – on revient sur les règles de covariance la semaine prochaine), alors c'est un ... overwrite.
  - La méthode réécrite dans la classe fille masque celle de la classe mère.
  - C'est assez naturel quand les paramètres sont identiques dans les deux méthodes, et bien plus étrange quand les paramètres sont différents. Mais il faut se souvenir qu'il n'y a jamais d'overload intergénérationel, c'est comme ça.

	Signature	**Scope**	**Comportement**
`override`	Inchangée	Mère-fille	Remplacement
`overload`	Modifiée	Même classe	Co-existence
`overwrite`	Inchangée mais méthode pas `virtual` dans mère ou bien signature modifiée (même si marquée `virtual`)	Mère-fille	Remplacement

II.2.d) Classe abstraite

- si une méthode est marquée `virtual =0` (méthode virtuelle pure), la classe ne fourni pas d'implémentation de cette méthode
- Cette classe est impossible à instancier (car on ne saurait pas quoi faire si quelqu'un appelle cette méthode sur cet objet). On dit que la classe est **abstraite**
- Les filles doivent implémenter cette méthode, sous peine d'être abstraites elles aussi
- C'est parfois utile pour factoriser du code: les autres méthodes de la classe abstraite peuvent appeler la méthode qui sera définie dans les filles, sans se préoccuper de comment c'est fait.
  - Exemple: définir `operator==(A& other)` en fonction de `operator!=(A& other)`

II.2.e) Implémentation du C++

- l'édition de liens d'une méthode virtuelle ne peut se faire à compil time ni meme à link time. Il faut attendre runtime car on ne connait pas le type dynamique de l'objet avant ce moment.
  - la classe a un champ magique `vtable` "table des méthodes virtuelles disponibles", et pour chaque méthode virtuelle, le compilateur génère un petit stub qui consulte cette table à l'exécution pour choisir la méthode à invoquer aujourd'hui.
  - le dtor doit être virtuel lui aussi, pour nettoyer la table des méthodes

II.3) Types objets en C++

II.3.a) Types statique et dynamique

- Le **type statique** (celui de la variable contenant l'objet) indique la liste des méthodes invocables sur l'objet (liste de signatures)
  - Connu dès la compilation
  - On peut le changer avec un transtypage
- Le **type dynamique** (celui donné à la création de l'objet) détermine la méthode qui sera invoquée
  - Connu uniquement à l'exécution, impossible à déterminer dans le cas général
- Souvent, c'est le même type, mais pas avec `A* a = new B();`
- Il s'agit bien d'une forme d'édition de lien à runtime, appellée **Liaison tardive (late binding)**

II.3.b) Transtypage

- Le type statique est le type donné à la variable. Le type dynamique est le type de la donnée stockée dans la variable
- `static_cast<A>(b)` équivalent du transtypage C: je te dis que c'est ce type, peu importe ce que tu penses, fais moi confiance
- `dynamic_cast<A>(b)` vérifie que c'est possible, s'il te plait.
  - Si oui, donne moi un objet du nouveau type (en utilisant les liens d'héritage, ou les opérateurs de convertion – cf. code fourni).
  - Si non, donne moi `nullptr`. Erreur à la compilation si c'est impossible d'après le type statique
  - Utilisable uniquement sur les pointeurs vers des objets, donc, pour que nullptr soit une valeur légale. Dans un langage moderne on aurait un type optionnel avec Some/None
  - c'est ce qu'on utilise pour savoir si l'objet contenu dans x est d'un type compatible au sens de Liskov avec la classe A (utilisable comme un A)

    `if (dynamic_cast<A>(x) != nullptr)` ou bien `if (auto* ax = dynamic_cast<A>(x)`
- `const_cast<A>(b)` juste pour changer la const-ness
- `reinterpret_cast<A>(b)` truc étrange rarement utile, sauf pour convertir entiers en pointeurs (donc, rarement utile)
- Le transtypage C est encore valide. Le compilo tente un `const_cast`, puis `static_cast`, puis `static_cast<const_cast>` avant des choses plus étranges.
- On regarde le code "Opérateurs de convertion" et on complète ce qui se passe dans chaque cas

```
1 struct A {
2     A(int) { }       // converting constructor
```

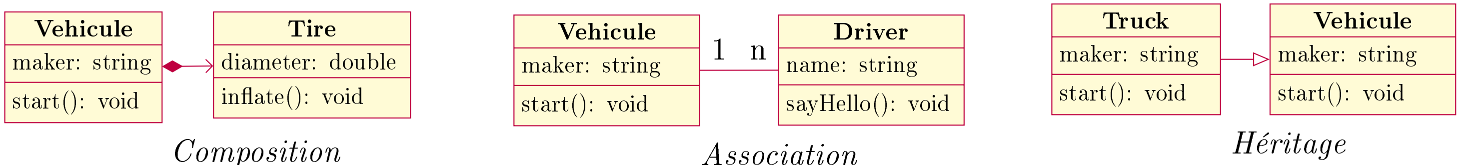
```
    A(int, int) { } // converting constructor (C++11)
    operator bool() const { return true; }
};
int main() {
    A a1 = 1;      // copy-initialization -> A::A(int) (sauf si explicit car '=')
    A a2(2);       // direct-initialization -> A::A(int)
    A a3 {4, 5};   // direct-list-initialization -> A::A(int, int)
    A a4 = {4, 5}; // copy-list-initialization -> A::A(int, int) (sauf si explicit)
    A a5 = (A)1;   // explicit cast performs static_cast
    if (a1) ;      // operator bool()
    bool na1 = a1; // copy-initialization -> A::operator bool() (sauf si explicit)
    bool na2 = static_cast<bool>(a1); // static_cast performs direct-initialization
}
```

III) Design OOP

- Quand je cherche à décomposer un problème de façon OOP, il faut donc que je trouve un ensemble de concepts qui se combinent bien
- Le plan d'ensemble est un schéma UML: des boites avec des flèches entre les boîtes
  - Chaque boîte est un concept (une classe), bien encapsulé
  - Chaque flèche dénote une interaction conceptuelle entre les concept
    - Relation 1: composition (**a-des**) quand on décompose un objet en sous-objets
    - Relation 2: association = composition réciproque: mise en relation de concepts
    - Relation 3: héritage (**est-un**) quand on voit une spécialisation d'un objet

III.1) Formalisme UML

- Vocabulaire graphique universel en matière de décomposition



- C'est ici une forme extrêmement simplifiée.
  - La visibilité publique/protected/privée peut s'écrire `+/-/\#`
  - La composition rend propriétaire des bouts; l'agregation (losange vide) non
  - `https://programmation-orientee-objet.pages.ensimag.fr/poo/resources/fiches/07-UML/`

III.2) Programmer proprement

- en général, on fait des sous-classes pour factoriser du code entre des trucs semblables. Mais c'est vite compliqué, les clients ne devraient pas forcément voir l'arborescence.
- Les classes abstraites sont pour proposer une interface à l'utilisateur, qui ne voit pas les implémentations
- L'héritage multiple est très dangereux, à proscrire sauf exception
- La surcharge de méthode et le sous-typage des paramètres ne font pas bon ménage, à proscrire

IV) Exception

- try/catch comme en Java. Pas de finally
- Les exceptions levées devraient être sous-classes de std::exception, mais ce n'est pas une obligation. On peut même lever des scalaires si on est sans coeur.
- On peut avoir plusieurs catchs d'affilé, le premier qui match est choisi.
- On lève une exception sans new (elle est copiée où il faut, pas besoin de chercher à faire fuiter sa mémoire) throw std::exception("Ouille");
- On catch par référence } catch (std::exception& e) { std::cerr « e.what(); }
- Quelques exceptions standards:
  - `logic_error`
    * `invalid_argument`: stoi("aze")
    * `domain_error`: std::acos(42)
    * `length_error`: on essaye de faire grandir un objet de taille statique
    * `out_of_range`: on lit en dehors
  - `runtime_error`
    * `range_error`: result cannot be represented by the destination type
    * `overflow_error` ; `underflow_error` ; `system_error` (C++11)
  - `bad_alloc`: erreur malloc
  - `bad_typeid`: typeid(nullptr)
  - `bad_cast`: `dynamic_cast` sans de lien d'héritage entre cible et type dynamique
  - `bad_exception`; `bad_weak_ptr`(C++11); `bad_function_call`(C++11)