Table des matières

Relations et structures **7**

Mise en garde ***8***

À propos de la forme ***9***

Héritage **10**

Option 0 : rédaction indépendante des trois classes ***11***

Option 1 : conception par composition ***13***

Enrober et déléguer 15

Option 2 : application de l'héritage ***17***

Implanter l'héritage de `Personne` à `Eleve` 18

Syntaxe de l'héritage en C++ 19

Héritage et constructeurs 20

Quelques règles d'identité 24

Héritage et destructeurs 25

Héritage et affectation 26

En résumé ***27***

Empêcher la dérivation ***28***

Dans d'autres langages ***29***

Exercices – Série 00 .... ***39***

Réflexions 40

Catégories d'héritage **41**

Spécifications d'accès et héritage ***41***

Héritage et consommation d'espace 42

Optimisation `EBCO` 43

Héritage public 44

Héritage public et membres publics 45

Héritage public et membres privés 46

Héritage public et membres protégés 47

Notes philosophiques sur l'héritage public 48

Exercices – Série `01.0` 49

Héritage privé 55

Héritage privé et membres du parent 57

Héritage privé et enrobage 58
Notes philosophiques sur l'héritage privé 60
Exercices – Série 01.1 61
Héritage protégé 67
Résumé des spécifications de sécurité et d'héritage 68
***Dans d'autres langages* 69**
***Mieux exploiter le code existant* 70**
***Héritage, surcharge de méthodes et espaces nommés* 72**
Ajouter à un espace nommé 73
L'espace nommé anonyme 73
Surcharge et dissimulation de noms 74
***Dans d'autres langages* 77**
Classes partielles 80
Méthodes d'extension 81
***Regroupements et notation UML* 82**
***Note technique sur class et struct* 83**
***Exercices – Série 02* 84**
Réflexion 88
Polymorphisme 89
***Retour sur la surcharge de méthodes* 90**
***Introduction au polymorphisme* 96**
Bref aperçu 96
Exemple concret 97
Polymorphisme et indirections 98
Polymorphisme et surcharge 101
Polymorphisme et contrats 103
Domaines d'application 104
Coûts du polymorphisme 104
Classes concrètes 104
Polymorphisme et opérateurs 106
Polymorphisme et constructeurs 107
Polymorphisme et vitesse 107

Classes terminales 108
Véritables cas de polymorphisme 108
Exercice – Série 03 .... ***110***
Réflexion 110
Le mot clé `virtual` ***111***
Dans le cas d'un destructeur 111
Mot clé contextuel `override` 113
Dans d'autres langages ***114***
Abstraction **117**
Méthodes abstraites ***117***
Classes abstraites ***118***
Interfaces 119
Rôle de l'abstraction ***120***
Polymorphisme en situation d'instabilité ***121***
Méthodes abstraites implémentées ***124***
Destructeurs abstraits 125
Dans d'autres langages ***126***
Exercices – Série 04 .... ***131***
Encadrer le polymorphisme et l'abstraction ***133***
Méthodes modèles (idiome `NVI`) 133
Héritage et paramètres par défaut ***135***
Héritage et classes concrètes ***136***
Exposer les membres d'un parent et clauses `using` ***138***
Clause `using` et constructeurs 139
L'héritage comme politique ***140***
Hériter pour ajouter 140
Hériter pour spécialiser 140
Hériter pour contraindre 141
Idiome de la classe `Incopiable` 141
Fonctions supprimées ou par défaut 143
Dans d'autres langages ***145***
Conversions explicites de types **147**

Problèmes du repérage de la conversion ***149***

Opérateurs de transtypage ISO ***150***

Conversion à la compilation : static_cast ***151***

Légalité statique et validité dynamique 152

Conversion à l'exécution : dynamic_cast ***154***

Validité dynamique et validation dynamique 155

En résumé 156

Dans d'autres langages ***157***

Exercices – Série 05 ..... ***160***

Autres opérateurs de transtypage ISO ***161***

Fermer les yeux : `reinterpret_cast` 161

Le type `void*` 163

Tricher sur la protection : `const_cast` 164

La qualification `mutable` 165

En résumé 167

Dans d'autres langages ***168***

Héritage multiple **169**

Arguments contre l'héritage multiple ***169***

De la relation entre héritage multiple et vitesse 170

Du caractère superflu de l'héritage multiple 170

De la relation entre héritage multiple et espace 171

De la relation entre héritage multiple et complexité 171

L'idée ***172***

La syntaxe ***174***

Héritage multiple et accès aux membres des parents 176

Héritage virtuel ***179***

Héritage virtuel et cycle de vie 183

Dénouer le nœud Gordien 184

Ramifications de l'héritage virtuel 185

Héritage multiple et affectation par défaut 185

Applications de l'héritage multiple ***186***

Héritage d'implémentation 187

Héritage d'interfaces ... 188
Hiérarchies planes ... 189
Combiner les stratégies ... 189
Parenthèse contemporaine ... 191
En résumé ... 192
Dans d'autres langages ... ***193***
Simuler l'héritage multiple ... ***197***
Classes imbriquées ... **198**
Exemple d'utilisation de classe imbriquée ... ***199***
Limites du secret ... ***203***
Dans d'autres langages ... ***204***
Techniques de clonage ... **205**
Duplication et indirections ... ***205***
Les clés d'un bon clonage ... ***208***
Le choix de la racine ... 208
Covariance : spécialisation des types des méthodes polymorphiques ... 210
Copier ou cloner? ... ***211***
Distinguer copie et clonage ... 211
Stratégie de duplication et nature des objets ... 212
Remarques supplémentaires sur la duplication d'objets ... ***213***
Dans d'autres langages ... ***214***
Nécessité et habitude ... 215
Exploiter l'idiome `RAII` ... **216**
Solution plus générale ... ***220***
Dans d'autres langages ... ***221***
Implanter un traitement d'exceptions rigoureux en C++ ... **222**
Facteurs de rigueur : stabilité et neutralité ... ***222***
Types standards d'exceptions ... 225
Manières d'attraper et stratégies connexes ... ***226***
Exceptions et vie des objets ... ***227***
Mécanique de dépilage automatique (Stack Unwinding) ... 228
Sous-programmes prudents ... ***228***
Les cas qui nous échappent ... ***231***

La fonction `std::uncaught_exception()` 231
La fonction `std::unexpected()` 231
La fonction `std::exit()` 232
La fonction `std::abort()` 232
La fonction `std::terminate()` 232
Dans d'autres langages ***233***
Exercices – Série 06 .... ***234***
Appendice 00 – Association, agrégation et composition **235**
Composition ***235***
Agrégation ***236***
Agrégation composition et délégation 238
Association ***239***
Appendice 01 – De l'importance des constantes dans le modèle OO **240**
Définition de l'encapsulation ***241***
Cas d'espèce ***242***
Vision C++ : constantes sur une base instance ***244***
Réaliser la constance d'un objet 246
Vision Java et .NET : immuabilité, ou constantes sur une base classe ***250***
Stratégies de programmation défensive 253
Immuabilité 254
Annexe 00 – Résumé de la notation UML abordée dans ce document **257**
Annexe 01 – Discussions sur quelques réflexions choisies **259**
Réflexion 01.0 : le polymorphisme est subjectif ***259***
Réflexion 01.1 : annoncer les exceptions ***260***
Réflexion 01.2 : exceptions sans effets secondaires ***261***
Annexe 02 – Covariance et contravariance (bref) **262**
Intension et extension ***263***
Individus **264**
Références **265**

Relations et structures

Pourquoi ce volume?

Ayant couvert dans [POOv00] ce qu'est un objet, avec les nuances entre instance et classe, et ayant travaillé à bien saisir le principe fondamental qu'est celui d'encapsulation, nous toucherons maintenant les autres grands thèmes de l'approche objet, en vue de hiérarchiser nos objets, de les organiser, de profiter des possibilités d'abstraction du modèle objet, de profiter de mécanismes de généralisation puissants, de dynamiser notre utilisation de l'approche, *etc*.

Ce volume couvrira l'héritage, le polymorphisme et l'abstraction, de même que la mécanique associée à ces idées, principalement en C++ mais avec des incursions dans d'autres langages OO pragmatiques comme Java, C# et VB.NET.

On y verra, entre autres choses :

- ce qu'est ***l'héritage***, qui s'exprimera entre autres parfois
  - sous la forme comment dériver une classe à partir d'une autre;
  - sous la forme comment spécialiser une classe;
  - sous la forme comment rassembler, à des fins opérationnelles et structurelles, les points en commun de plusieurs classes en un même lieu;
  - sous la forme *comment restreindre les possibilités associées à un groupe de classes* (ce qui, croyez-le ou non, est aussi utile que d'étendre la gamme des possibilités qui leur sont offertes); *etc*.
- les catégories d'héritage;
- les relations entre un ***descendant*** et ses ***ancêtres***;
- ce qu'est une méthode abstraite;
- ce qu'est une classe abstraite;
- ce que sont les ***interfaces***, le terme étant pris dans un sens spécialisé;
- les méthodes virtuelles et le polymorphisme;
- la ***vérification des types*** à la compilation et à l'exécution;
- l'héritage multiple et l'héritage virtuel; et
- les relations entre classes que sont l'***association***, la ***composition*** et l'***agrégation***.

Nous profiterons de ces nouveaux acquis pour enrichir notre compréhension de plusieurs des idées couvertes dans [POOv00] en particulier celles ayant traits aux exceptions et à la duplication des objets.

Mise en garde

La présente couvrira les *points techniques* relatifs à l'héritage, au polymorphisme et à l'abstraction, l'objectif derrière cette série de documents demeurant celui de développer chez l'étudiant(e) une connaissance opérationnelle de la POO.

On touchera aussi à l'occasion à des *éléments de design*, puisqu'il nous faudra, en plus de développer des aptitudes techniques, aiguiser notre regard et développer une approche qui nous permettra de choisir une solution viable lorsque nous ferons face à un problème de conception.

Pourquoi ces nouvelles idées?

En effet, là où le principe d'encapsulation voit chaque classe comme une entité indépendante et qui cherche à garantir le respect pour elle-même d'un certain nombre de règles, l'héritage et le polymorphisme nous proposent de concevoir nos classes de manière hiérarchique : de réfléchir à leur organisation, à leurs interrelations et à nous poser des questions quant aux points en commun qu'ont entre elles certaines classes.

Ceci nous amènera à examiner en quoi nous pouvons tirer profit, au plan opérationnel, de ces traits qui leur sont propres.

Raisons derrière la séquence des acquis

Pour approcher certains problèmes de design, il faut au préalable connaître le vocabulaire et les bases du langage dans lequel on s'exprime. C'est en partie pourquoi ce cours repose sur une compréhension des bases de la programmation structurée, et pourquoi nous avons d'abord procédé à un examen appliqué de ce que sont les classes, les instances et le principe d'encapsulation.

Prise de décisions

Les décisions de design sont celles où seront surtout impliqués les analystes et les chefs de projet. Dans une situation de prise de décision sur une question de design, en comprenant les bases sur lesquelles sont construites (ou seront construites) les fondations des projets conçus ou maintenus par/ dans leur entreprise, ces individus seront appelés à trancher parmi les différentes options qui se présenteront à leur équipe. Quand plusieurs solutions viables se présenteront à eux, il leur faudra savoir discriminer et identifier les solutions à privilégier, pour l'entreprise, le projet et l'équipe de développement.

Les décisions prises par ces personnes auront un impact philosophique et technique dont les répercussions se feront sentir à long terme. Bien qu'une personne appelée à prendre des décisions risque peu d'être impliquée à plein temps dans la partie *codage* du projet, elle devra être assez *crédible* aux yeux de l'équipe technique pour que celle-ci accepte ses idées et sa vision, et être assez éveillée face aux différents enjeux techniques et politiques liés au développement pour que sa vision mène vers un produit rentable et surtout de qualité.

À propos de la forme

Ce document, bien que plus sophistiqué que son prédécesseur [POOv00], demeure un document explorant les bases du modèle OO. De ce fait, il contient de remarques qui mériteront d'être raffinées et des pratiques sur lesquelles nous reviendrons ultérieurement. Prenez donc soin de garder l'esprit ouvert puisque ce qui convient pour démarrer un processus de réflexion peut ne pas convenir à une pensée plus avancée sur un même sujet.

Vous trouverez à divers endroits des petits encadrés indiquant ***Réflexion 01.n*** (pour divers entiers ***n***). Ces encadrés soulèveront des questions qui méritent une discussion plus approfondie et pour lesquelles les réponses peuvent surprendre ou ne pas être aussi banales qu'il y paraît. Des ébauches de réponses seront proposées pour chacune dans la section ***Annexe 01 – Discussions sur quelques réflexions*** choisies de ce document.

Puisque ces notes se veulent un appui à l'apprentissage de la POO, pas d'un langage particulier, mais puisqu'il faut aussi utiliser (au moins) un langage pour appuyer formellement nos idées et notre discours, le cœur de ce document utilise un langage OO, C++, pour ses exemples. Cependant, vous trouverez à certains endroits dans le document des sections intitulées ***Dans d'autres langages*** qui exploreront les ramifications des sections précédentes en Java, C# et VB.NET, ce qui vous permettra de faire le pont avec d'autres technologies. Ces pages ont des bordures différentes des autres, et vous pourrez les omettre si ces nuances ne sont pas au cœur de vos préoccupations.

Enfin, notez qu'au moment d'écrire ces lignes, la norme la plus récente du langage C++ est **C++ 14**, un ajustement à l'énorme mise à jour que fut C++ 11, le vote pour officialiser C++ 17 (la prochaine mise à jour significative du langage) est en cours, et les travaux sur ce qui devrait être C++ 20 vont bon train. Les volumes plus poussés de cette série de notes de cours couvrent des aspects des plus récentes versions de C++, et montrant parfois comment il est possible d'en arriver au même résultat à partir de versions antérieures du langage, ou en indiquant les raisons qui motivent certaines adaptations de la norme.

Héritage

Pourquoi cette section?

Nous allons voir ici les grandes lignes de la raison d'être de l'héritage dans une approche objet, en examinant les alternatives possibles et en mettant en relief les situations dans lesquelles on préférera l'une ou l'autre des approches possibles.

Présumons un modèle `OO` simplifié à l'intérieur duquel auraient été développées les trois classes suivantes[1] :

- la classe `Eleve`, dont chaque instance représentera un élève[2];
- la classe `Prof`, dont chaque instance représentera un professeur; et
- la classe `Client`, dont chaque instance représentera un client pour une firme commerciale quelconque.

On pourrait vouloir connaître :

- d'un *élève* : nom, prénom, sexe, les cours suivis (des objets en soi?) et ses résultats scolaires;
- d'un *professeur* : nom, prénom, sexe et la liste des cours qu'il enseigne;
- d'un *client* : nom, prénom, sexe, adresse postale et le montant de ses comptes en souffrance.

Pour nos fins, nous présumerons l'existence d'un type `Genre` définissant les catégories de sexe possibles pour une instance de `Personne` donnée. Une implémentation possible du type `Genre` serait l'énumération proposée à droite. Cela suffira pour nos fins[3].

```
enum Genre {
   Feminin, Masculin
};
```

Même dans ce modèle simplifié, on constatera que nos trois classes ont des points en commun : elles représentent toutes des entités ayant un *nom*, un *prénom* et un *sexe*[4]. Cette communauté structurelle est un signe que leurs implémentations *peuvent* être liées d'une certaine façon.

---

[1] Vous pouvez sans peine vous amuser à les définir vous-mêmes, à titre d'exercice!

[2] J'ai choisi élève plutôt qu'étudiante ou étudiant du fait que le terme élève est épicène.

[3] Dans ***Exercices – Série 00***, plus loin, l'exercice `EX04` vous propose une autre manière de conceptualiser le type `Genre`. Il s'agit d'un exercice intéressant à plusieurs égards.

[4] **Attention** : il n'est pas clair que chaque entité a précisément ces trois attributs (quoique ce puisse être le cas), mais bien que chacune offrira des opérations donnant accès à ces informations. Sur le plan de l'implémentation, on pourrait trouver :

- deux `std::string` (nom et prénom) et un `bool` (pour le sexe);
- deux `std::string` et un `short` ou un `enum`, surtout si, dans notre monde diversifié, on envisage offrir plus de deux choix pour la mention *sexe*;
- on pourrait aussi penser à une seule instance de `std::string` dans laquelle on séparerait le nom, le prénom et le sexe par des délimiteurs (p. ex. : `"Roger;Tromblon;M"`);
- à la limite, on pourrait ne garder qu'une valeur (un numéro d'assurance sociale, peut-être) identifiant la personne dans un registre plus complet (une base de données, par exemple) et par laquelle on peut obtenir ces informations au besoin.

Ici, en fait, un constat évident se dessine : les trois classes proposées ont en commun de représenter des cas particuliers de personne. Dans un cas comme celui-ci, trois grandes options (au moins[5]) s'offrent à nous pour le design de nos classes. Nous connaissons déjà les deux premières, bien que nous soyons pour la première fois sur le point de leur donner un nom, mais c'est la troisième que nous privilégierons.

Les classes utilisées pour fins d'exemple dans cette section utilisent des accesseurs, des mutateurs, des opérations de la Sainte-Trinité écrites à la main, etc. Ces choix d'implémentation ne sont pas idéaux pour la classe que nous examinerons; ils ne sont là que pour faciliter la discussion des concepts qui nous intéressent ici.

Allons-y donc d'un examen rapide des options en question.

Option 0 : rédaction indépendante des trois classes

La 1re option s'offrant à nous est celle de rédiger les trois classes de manière indépendante et disjointe l'une de l'autre. Nous pouvons d'ailleurs y arriver sans avoir recours à d'autres connaissances que celles déjà à notre disposition.

Dans ce cas :

- on rédigera probablement une première classe, puis on la testera;
- une fois la première classe opérationnelle, on procédera à une séance de copier/ coller pour débuter la rédaction de la seconde classe, qu'on complétera avec ce qui la différencie de la première[6].On vérifiera ensuite qu'elle se comporte comme désiré;
- on répétera une procédure semblable pour la classe suivante.

Cette façon de faire implique un certain nombre de manipulations, mais est en général faisable et accessible à tous sur le plan technique comme sur le plan conceptuel.

On obtient ainsi trois classes *similaires aux yeux des analystes humains* comme nous, qui sommes capables de voir les similitudes structurelles et opérationnelles en regardant le code ou la description de chacune, mais *dissociées au sens du code*, puisque rien n'indique explicitement dans chacune des trois classes qu'elle contient des éléments similaires aux deux autres, et de quelle façon.

Un compilateur ne peut présumer de similitudes entre deux entités simplement parce qu'elles ont des données et des opérations similaires, même si les types et les noms de leurs membres concordent. Pour tirer de telles conclusions, il faut une connaissance du contexte, donc être capable de prendre position et d'affirmer la proximité de nature entre un `Prof` et un `Eleve` (et encore, on pourrait bien se tromper).

[5] On pourrait aussi discuter d'***agrégation***, qui est une autre technique importante, mais le temps nous manquerait et nous dévierions alors du sujet principal de la présente. Vous pouvez toutefois lire à ce sujet, entre autres dans Internet, cette technique étant parfois une option intéressante quand l'*héritage d'implémentation* n'est pas une option disponible. Voir ***Appendice 00 – Association, agrégation et*** composition pour des détails.

[6] Par exemple, si on a d'abord rédigé `Eleve` et qu'on rédige maintenant `Prof`, on ne gardera que ce qui est commun aux deux, retirant ce qui est spécifique à `Eleve`, et on ajoutera ensuite ce qui est spécifique à `Prof`.

```
Eleve
- nom_: std::string
- prenom_: std::string
- sexe_: Genre
...autres attributs d'un Eleve...
+ GetNom() const : std::string
+ SetNom(const std::string &): void
+ GetPrenom() const : std::string
+ SetPrenom(const std::string &): void
+ GetSexe() const : Genre
+ SetSexe(Genre): void
+ EstMasculin() const : bool
+ EstFeminin() const : bool
...autres méthodes d'un Eleve...
```

```
Prof
- nom_: std::string
- prenom_: std::string
- sexe_: Genre
...autres attributs d'un Prof...
+ GetNom() const : std::string
+ SetNom(const std::string &): void
+ GetPrenom() const : std::string
+ SetPrenom(const std::string &): void
+ GetSexe() const : Genre
+ SetSexe(Genre): void
+ EstMasculin() const : bool
+ EstFeminin() const : bool
...autres méthodes d'un Prof...
```

```
Client
- nom_: std::string
- prenom_: std::string
- sexe_: Genre
...autres attributs d'un Client...
+ GetNom() const : std::string
+ SetNom(const std::string &): void
+ GetPrenom() const : std::string
+ SetPrenom(const std::string &): void
+ GetSexe() const : Genre
+ SetSexe(Genre): void
+ EstMasculin() const : bool
+ EstFeminin() const : bool
...autres méthodes d'un Client...
```

On verra plus loin que, si cette option est viable à court terme, elle demande de réécrire plusieurs fois la même chose[7] et nous prive d'un outil extrêmement puissant (le *polymorphisme*) qui dépend de l'existence d'un lien explicite de parenté entre les classes.

> ***Andrew Hunt*** et ***Dave Thomas***, dans [PragProg], mettent de l'avant ce qu'ils nomment le principe `DRY`, pour *Don't Repeat Yourself*.
>
> Le copier/ coller est une violation directe du principe `DRY`. Bien qu'il permette à court terme d'atteindre un objectif, fait en sorte que toute modification ultérieure à l'un des éléments ayant au préalable été copiés puis collés devra être réalisée à plusieurs endroits. Ceci accroît les manipulations à réaliser, augmente les risques d'erreurs, et est contreproductif de manière générale.

À droite, dans un schéma respectant la norme `UML`, on voit que les trois classes sont logiquement dissociées, malgré leurs similitudes structurelles et opérationnelles. Évidemment, le schéma ne trace que les grandes lignes de chaque classe, comme le permet d'ailleurs `UML`. On ajouterait normalement une pléthore de méthodes additionnelles, par exemple des opérateurs de comparaison et d'affectation, des constructeurs, *etc*.

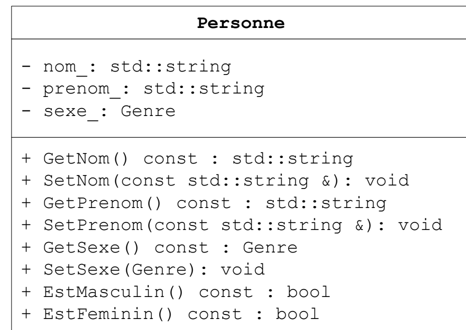
> **Réflexion** : bien que ces classes soient présentées avec une gamme symétrique d'accesseurs et de mutateurs, en pratique, implémenteriez-vous ces opérations, en tout ou en partie? Si oui, lesquelles et pourquoi? Sinon, pour quelle raison vous abstiendriez-vous?
>
> Notez qu'il n'y a pas de réponse universellement adéquate à cette question, mais la réflexion qui l'accompagne est féconde.

[7] Bien que le copier/ coller facilite de telles opérations de duplication, il demeure que plus on doit réaliser de manipulations *manuelles*, plus grand devient le risque d'erreurs. Sans compter que si on doit maintenir à jour deux (ou plus) versions différentes du même code, la maintenance de l'un et de l'autre se complique grandement.

Option 1 : conception par composition

L'approche par composition reconnaît la similitude entre les trois classes, mais l'explique en spécifiant que *dans chaque* Eleve[8], *il y a une* Personne.

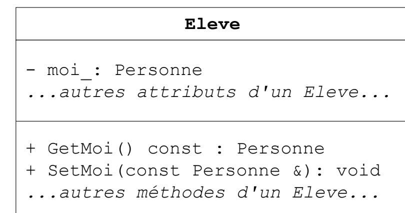
La **composition** implantera ce constat en créant une classe Personne, qui aura une vie indépendante de celle de la classe Eleve[9].



L'introduction d'une classe supplémentaire constitue un gain très net sur la technique précédente, qui reposait sur la duplication manuelle du code source. En effet, en identifiant l'idée de Personne, commune aux trois autres classes, et en lui donnant une existence en propre, on obtient tout d'un coup une entité qui pourra être utilisée dans d'autres contextes.

Dans la classe Personne, ci-dessus, on retrouvera les attributs et méthodes que toute Personne devrait avoir, du moins selon notre conception (simpliste) de cette classe.

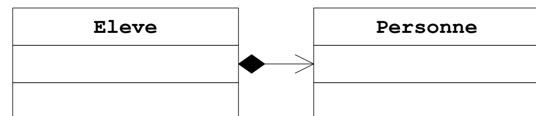
La classe Eleve, quant à elle, aura un attribut qui sera une instance de Personne, et qui offrira déjà tout ce qu'une Personne devrait offrir. Puisqu'il faut lui donner un nom, nous l'appellerons pour le moment moi_[10].



Pour permettre à des entités externes un accès contrôlé à cet attribut, la classe Eleve pourra présenter un accesseur et un mutateur publics, disons GetMoi() et SetMoi().

La méthode GetMoi() retournera un Personne, un Personne& ou un const Personne&[11], selon le type d'accès désiré. La pertinence du SetMoi() est matière à débats, mais cela importe peu pour cette discussion.

Rappelons au passage la notation UML pour décrire une relation de composition dans laquelle tout Eleve contient une Personne.



[8] …et chaque Prof, et chaque Client… nous limiterons notre explication au cas de la classe Eleve, pour alléger le texte.

[9] On pourra donc, au besoin, instancier Personne sans nécessairement avoir besoin de construire un Eleve, un Prof ou un Client pour y arriver. Ceci n'est pas une évidence, ou quelque chose d'immédiat, car il existe des classes qu'on ne peut pas instancier. Ces classes sont dites *abstraites*, et nous discuterons d'elles plus loin.

[10] Ceci est un peu boiteux, mais il y a une raison pour laquelle il est difficile ici de donner un nom convenable à cet attribut – raison qui deviendra apparente lorsque nous discuterons de la 3e option.

[11] C'est une question de design et de sécurité, qui dépend de plusieurs facteurs, la plupart dépendant du contexte de développement et d'utilisation des objets. On ne peut y répondre brièvement et de manière vraiment générale.

L'approche par composition fonctionne[12], quoiqu'elle comporte certains détails agaçants dans le cas qui nous intéresse ici. Par exemple, si la méthode GetMoi() retourne par copie une instance de Personne et s'il s'avère qu'une entité externe désire modifier le nom d'un Eleve, il lui faudra :

- en premier lieu, créer une nouvelle Personne qui sera une copie de celle contenue dans l'instance de Eleve qui est visée par cette opération;
- modifier le nom de cette nouvelle Personne; et
- réaffecter la nouvelle Personne à l'instance d'Eleve.

```
// si Eleve::GetNom() est
// de type "Personne"
Eleve e;
// ...
Personne p{e.GetMoi()};
p.SetNom("Tromblon");
e.SetMoi(p);
```

Cela représente une masse plus imposante de calculs qu'il n'y paraît, et n'est pas tout à fait ce qu'on qualifierait habituellement d'*élégant*.

On pourrait éviter une partie du problème en retournant une référence à la Personne dans l'instance d'Eleve plutôt qu'une copie de cette Personne (ce qui est un choix en partie risqué[13]). Même avec cette approche, pour modifier le nom d'un Eleve, il faudrait passer par une étape intermédiaire : la Personne encapsulée à l'intérieur. *On perd, dans la notation, l'idée qu'on cherche à référer au nom de l'élève*.

```
// si Eleve::GetNom() est de
// type "Personne&"
Eleve e;
// ...
e.GetMoi().SetNom("Tromblon");
```

Cette situation est assez agaçante pour qu'on ait parfois tendance à enrober (par des *Wrappers*) les méthodes des objets internes (comme moi_, notre Personne) avec des méthodes de l'objet qui le contient.

---

[12] En fait, on utilise à peu près toujours un peu de composition dans la conception d'une classe, quand bien même ce ne serait que pour spécifier les attributs de ladite classe. Ce qu'il y a de particulier ici, c'est la nature de la relation particulière entre les idées de Personne et de Eleve (ou entre Personne et Prof, ou …). Selon vous, en quoi est-ce là une relation différente de celle entre un Rectangle et un Point lui servant d'origine?

[13] Retourner une référence sur la Personne dans un Eleve serait-il un bris d'encapsulation? Envisagez ce cas : imaginons que les règles selon lesquelles un nom de Personne est valide varient selon d'une classe à l'autre. Si Eleve et Client ont des règles de validation distinctes, alors ces classes voudront valider tout changement au nom, donc prendre en charge et encadrer elles-mêmes les modifications à cet état. Si Eleve expose une référence directement sur son moi_ intérieur, cela permettra au code client de modifier directement la Personne dans un Eleve, donc de passer outre les politiques de validation de la classe Eleve. Toute instance de la classe Eleve perdrait donc son propre contrôle sur ses états : l'encapsulation serait brisée. Une solution à de tels irritants est ce qu'on nomme la ***Programmation par politiques*** [hdPol].

Ce type de bris d'encapsulation est très répandu dans les langages comme Java et les langages .NET, du fait que ces langages manipulent tous les objets par des indirections; bien que ce choix philosophique soit raisonnable, trop peu de programmeurs utilisant ces langages pour s'exprimer prennent véritablement conscience de l'impact de cette réalité sur leur manière de programmer. Si vous utilisez ces langages, soyez très prudent(e)s face aux bris d'encapsulation silencieux que sont ces problèmes d'*aliasing*, et favorisez les classes immuables chaque fois que cela s'avère possible et raisonnable. Pour en savoir plus sur le sujet, voir [JavaAlias].

Enrober et déléguer

La composition, même s'il ne s'agit pas de la meilleure approche ici, est une technique importante, répandue et, dans bien des cas, sera effectivement l'approche à privilégier pour représenter une relation entre deux objets (par exemple, la relation entre une instance de `Personne` et son nom).

Présumant que nous souhaitions procéder par composition dans ce cas-ci, nous voudrions probablement faciliter des opérations telles que consulter ou modifier le nom d'un `Eleve`, à la fois pour que ces opérations soient plus naturelles (qu'elles semblent faire directement partie de l'interface de la classe `Eleve`) et pour permettre à la classe *englobante* (`Eleve`) d'appliquer certaines de ses règles de validité sans pour autant devoir implémenter en totalité les méthodes de la classe *englobée* (`Personne`).

Typiquement, on réalise cette tâche par ***enrobage*** et ***délégation***. Dans le cas qui nous intéresse, `Eleve` enrobera `Personne` et le fera disparaître des yeux du code client. Par la suite, `Eleve` implémentera des versions « bidon » des méthodes de `Personne` qu'il souhaite exposer (peut-être toutes ces méthodes, peut-être un sous-ensemble de celles-ci). Dans sa déclinaison la plus élémentaire, la version de la classe *englobante* sera ce qu'on nomme un *Pass-Through*, une délégation telle quelle, vers la version de la classe englobée.

```
// ... inclusions etc.
class Eleve {
   Personne moi_;
   // ...
public:
   void SetNom(const string &nom) {
      moi_.SetNom(nom);
   }
   // ...
};
```

Au besoin, `Eleve` pourra appliquer ses propres politiques de validation, en particulier sur les intrants de ses mutateurs.

Comme le montre l'exemple à droite, à l'aide du mutateur `SetNom()`, un `Eleve` pourrait recevoir la demande de changement de nom, appliquer ses politiques locales de validation, et déléguer à la `Personne` cachée à l'interne la tâche de procéder au changement de nom (dans le respect de ses propres politiques) seulement dans le cas où cela s'avère acceptable.

```
// ... inclusions etc.
class Eleve {
   Personne moi_;
   // ...
   static bool est_nom_valide(const string&);
public:
   class NomIllegal {};
   void SetNom(const string &nom) {
      if (!est_nom_valide(nom))
         throw NomIllegal{};
      moi_.SetNom(nom);
   }
   // ...
};
```

Évidemment, il importe dans un tel cas que les politiques de validité des deux classes impliquées comprennent une intersection non vide (dans le cas contraire, il faudra réviser le choix de faire collaborer ces classes, ou encore réévaluer les politiques de validité de l'une ou de l'autre).

Il est même possible de mettre en place des mécanismes protégeant en partie la classe *englobante* de décisions ne lui convenant pas de la part de la classe *englobée*.

Présumant une encapsulation stricte, dans l'exemple à droite, `Eleve` est en droit de supposer que l'intégrité de `Personne` est en tout temps respectée. Il se peut toutefois que le mutateur `SetNom()` de `Personne` procède à des ajustements à la valeur reçue en paramètre qui, elles, ne conviennent pas à `Eleve`.

En saisissant au préalable la valeur du nom avant invocation du mutateur de `Personne`, un `Eleve` peut (présumant que `Personne` ne soit pas une classe malicieuse) garantir sa propre cohérence et, au besoin, revenir à l'état initial suite à un changement d'état qui lui poserait problème. Évidemment, il s'agit de cas extrêmes; l'idéal, lorsque deux classes vivent un couplage serré, est de faire en sorte que leurs politiques de validation respectives concordent, ou à tout le moins qu'elles ne se nuisent pas mutuellement.

```
// ... inclusions etc.
class Eleve {
   Personne moi_;
   // ...
   static bool est_nom_valide(const string&);
public:
   class NomIllegal {};
   class Incoherence {};
   void SetNom(const string &nom) {
      auto avant = moi_.GetNom();
      if (!est_nom_valide(nom))
         throw NomIllegal{};
      moi_.SetNom(nom);
      if (!est_nom_valide(moi_.GetNom())) {
         moi_.SetNom(avant);
         throw Incoherence{};
      }
   }
   // ...
};
```

En résumé, enrober les méthodes d'une `Personne` avec les méthodes d'un `Eleve` donne l'illusion qu'`Eleve` implante ces méthodes alors qu'il ne fait que déléguer la responsabilité du traitement à un objet interne. Ceci faciliterait l'utilisation d'un `Eleve` en tant que `Eleve`.

Le défaut, bien sûr, est qu'il peut être pénible d'enrober chaque méthode d'un objet englobé par une méthode de l'objet englobant. En effet, si on modifie la classe `Personne`, chaque classe qui s'en sert par composition et veut en enrober les méthodes risque d'avoir à être réécrite, du moins en partie. Ce couplage complique évidemment l'entretien du code.

Petite remarque : un objet enrobé peut lui-même en enrober d'autres. Sachant cela, prenez soin (a) d'éviter les cycles, surtout si vous accédez indirectement aux objets englobés, et (b) d'être conscientisés face aux risques d'impacts d'un trop long enchaînement de délégations sur la performance d'ensemble des méthodes.

Les systèmes d'exploitation comme *Win32* ou `UNIX` dévoilent généralement une interface de programmation (`API`) permettant de communiquer par programmation avec le système. Une telle `API` est, à ce jour, presque toujours écrite selon le modèle structuré, le langage C servant de plus petit commun dénominateur dans la plupart des cas.

Présumons qu'on désire appliquer une approche `OO` à une fonctionnalité du système, par exemple le système de fichiers (c'est d'ailleurs ce que font les bibliothèques de classes de Java et des langages `.NET`). On peut alors se créer une classe représentant le système de fichiers, avec des méthodes pour créer un fichier, le renommer, *etc*.

Évidemment, ne voulant pas réinventer la roue, alors on fera en sorte que chaque méthode appelle, à l'interne, la fonction la plus appropriée de l'`API` du système… et on aura appliqué la technique de l'enrobage!

Option 2 : application de l'héritage

La composition est une technique efficace et qui a sa raison d'être. On procède d'ailleurs régulièrement par composition sans vraiment s'en apercevoir. Prenons seulement quelques exemples très simplifiés :

- une `Automobile` constitue un regroupement de quatre roues, un moteur, un châssis, des banquettes, et ainsi de suite;
- un `Nombre` peut être vu comme une suite organisée de chiffres;
- un `Eleve` détient, en quelque sorte, les notes et les cours qu'il a suivi;
- un `Costume` rassemble dans un tout cohérent plusieurs vêtements; *etc*.

L'une des raisons pour laquelle le schème de composition ne colle pas aussi bien à la relation entre `Eleve` et `Personne` qu'à celle entre une automobile et ses roues est que, là où l'automobile *contient* ou *possède*, entre autres, des roues, un `Eleve` *est* une `Personne`. En fait, tout dépendant de ce qu'on veut exprimer, on dira sans se tromper que :

- un `Eleve` est au moins une `Personne`;
- un `Eleve` est un cas particulier de `Personne`;
- un `Eleve` est une spécialisation de `Personne`;
- un `Eleve` est un dérivé de `Personne`; ou encore
- un `Eleve` est tout ce qu'une `Personne` est, *et même plus*[14]!

Chacun de ces énoncés repose sur l'emploi d'un même verbe…

La présence du verbe *être* dans chacune des affirmations ci-dessus, de même que la manière dont ce verbe est utilisé, n'est pas le fruit du hasard; l'analogie généalogique non plus, d'ailleurs. Lorsqu'une relation sur la base du verbe être se dessine, la technique de la composition d'objets est souvent moins appropriée que le recours à ce qu'on nomme l'***héritage***.

⇒ L'**héritage**, au sens d'*héritage simple*[15], permet de faire d'une classe, l'***enfant***, une spécialisation d'une autre classe, le ***parent***.

⇒ Si `E` hérite de `P`, alors `E` a intrinsèquement tout ce que `P` avait, *plus* ce qui distingue `E` de `P`.

Quelques synonymes (liste non exhaustive)…	
Pour ***héritage*** :	dérivation[16], *sous-classer* (*subclassing, to subclass*), spécialisation.
Pour ***enfant*** :	classe dérivée, descendant, sous-classe (*subclass*), cas particulier, spécialisation, héritier ou descendant (surtout s'il n'est pas enfant immédiat).
Pour ***parent*** :	classe de base, ancêtre, superclasse (*superclass*), généralisation, ancêtre (surtout s'il ne s'agit pas du parent immédiat).

[14] Le mot *plus* doit être pris ici au sens de *plus spécialisé*, ce qui peut entre autres signifier *possède plus d'attributs*, *dévoile des méthodes que* `Personne` *ne dévoilait pas*, ou encore *fait certaines opérations de manière plus raffinée, plus spécialisée que ne le faisait* `Personne`.

[15] Il existe aussi dans certains langages, dont C++, Eiffel et `CLOS` tout comme, dans la mesure où on se limite à l'héritage d'*interfaces*, Java et les langages `.NET`, la possibilité d'y aller d'héritage multiple. On y reviendra.

[16] À ne pas prendre au sens physique ou mathématique!

Implanter l'héritage de `Personne` à `Eleve`

Ce qui suit présente un exemple pas à pas d'héritage simple, pour aider à comprendre la syntaxe impliquée et le sens des symboles utilisés. Il s'agit d'un exemple simple, bien sûr.

Examinons concrètement ce qu'implique d'appliquer l'héritage entre `Personne` et `Eleve`, du moins en surface, à l'aide du langage C++. Aucun jugement de valeur n'est posé quant à la qualité de l'interface publique de `Personne` ou quant au bon usage de la Sainte-Trinité; nous ne visons qu'à montrer la mécanique derrière le concept à l'aide d'un exemple simple.

Tout d'abord, nous présumerons la classe `Personne` telle que présentée à droite (extrait de `Personne.h`).

Il est présumé que cette déclaration de classe vous apparaît claire au point où nous en sommes rendus. Les détails d'implémentation (par exemple : qu'est-ce qui fait que deux instances de `Personne` sont jugées pareilles ou différentes selon `==` ou `!=`) sont laissées à votre bon jugement, n'étant pas essentielles à notre discussion.

Dans le doute, essayez de rédiger la classe `Personne` en entier, en fonction de l'interface proposée par la déclaration à droite.

À ce stade-ci, on veut faire de la classe `Eleve` une classe qui ait tout ceci, en plus de ce qui fait d'un `Eleve` quelque chose de spécial par rapport à une `Personne` prise en un sens plus général.

```
#ifndef PERSONNE_H
#define PERSONNE_H
#include <string>
using std::string;
enum Genre { Feminin, Masculin };
class Personne {
public:
   // constructeurs, destructeur
   Personne();
   Personne(const string &nom,
            const string &prenom,
            Genre);
   Personne(const Personne&);
   ~Personne();
   // Accesseurs
   string GetNom() const;
   string GetPrenom() const;
   Genre GetSexe() const noexcept;
   // Mutateurs
   void SetNom(const string &);
   void SetPrenom(const string &);
   void SetSexe(Genre);
   // Opérateurs
   Personne& operator=(const Personne&);
   bool operator==(const Personne&) const;
   bool operator!=(const Personne&) const;
private:
   string prenom_,
          nom_;
   Genre sexe_;
};
#endif
```

Nos efforts dans le cadre de cette section iront donc dans le sens de rédiger correctement la classe `Eleve` en fonction de cette contrainte. Nous procéderons de manière progressive, ajoutant un élément à la fois.

Syntaxe de l'héritage en C++

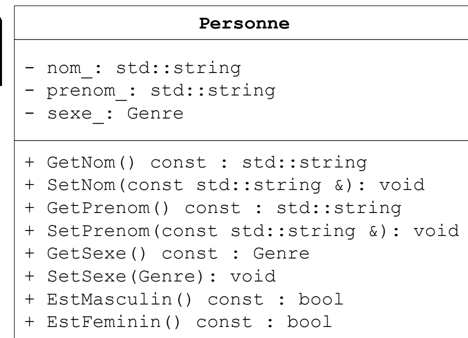
Pour décrire que la classe `Eleve` *hérite* de la classe `Personne` en C++, on rédigera tout d'abord du code tel que celui visible ci-dessous.

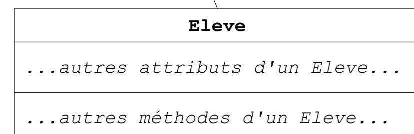
```
// ... dans Eleve.h ...
#include "Personne.h"
class Eleve
   : public Personne
{
   // ...
};
// ...
```

Quelques éléments sont à remarquer :

- toute chose, en C++, doit être déclarée avant d'être utilisée. Ainsi, dans le fichier `Eleve.h` ci-dessus, on inclura d'abord le fichier `Personne.h` pour avoir accès à la déclaration de la classe `Personne`. Une déclaration *a priori* ne suffira pas, puisque tout `Eleve` est une `Personne`, donc puisque la structure de `Eleve` comprend au moins celle de `Personne`;
- la manière d'indiquer que `Eleve` dérive de `Personne` est d'ajouter, à la suite de `class Eleve`, la mention **`: public Personne`**;
- le mot clé **`public`** a ici le sens de Oui, un `Eleve` est une `Personne`, et tout le monde le sait; **il s'agit d'un savoir public**.

> Il est possible d'utiliser d'autres mots clés que `public` dans cette situation. Nous y reviendrons.





En utilisant la notation `UML`, la relation d'héritage que nous venons de créer s'exprimerait comme dans le schéma présenté à droite. Nous aurions pu y aller avec un niveau moindre de détail et simplifier le schéma pour n'inclure que les noms de classes et la relation entre les deux classes impliquées, omettant les détails d'implémentation de l'une comme de l'autre.

L'expression graphique a le mérite de nous présenter clairement que nous n'avons pas, dans `Eleve`, à répéter ce qui est déjà écrit pour `Personne`. Hériter de `Personne` donne immédiatement à la classe `Eleve` tout[17] ce que l'on trouvait déjà dans `Personne`.

Une des principales différences entre une implantation par héritage et une autre par composition est que, si `Eleve` hérite publiquement de `Personne`, alors *toute instance de la classe `Eleve` est aussi une instance de la classe `Personne`*.

```
#include "Personne.h"
#include "Eleve.h"
void f(const Personne&);
int main() {
   Eleve e;
   e.SetNom("Tromblon");
   f(e);
   // ...
}
```

Ainsi, l'extrait de code proposé à droite, qui déclare une instance de `Eleve` et s'en sert en appelant une méthode publique de `Personne` ou en passant un `Eleve` à une méthode exigeant une `Personne`, est ***valide***. Un *`Eleve`* ***est****,* par héritage*, une `Personne`*.

[17] Prudence ici, car la généralisation est un peu hâtive et ne se veut pour le moment qu'une grossière illustration; nous y reviendrons sous peu.

Héritage et constructeurs

Sans vouloir nous avancer trop loin dans la technique pour le moment, nous allons quand même nous demander comment se combinent héritage et constructeurs.

En effet, si le rôle d'un constructeur est d'initialiser correctement un objet, on est en droit de se demander comment une instance d'une classe dérivée doit être construite. Après tout, il y a en elle tout ce que son parent contenait, plus ce qui la rend spéciale. Y a-t-il un ordre particulier pour procéder à la construction?

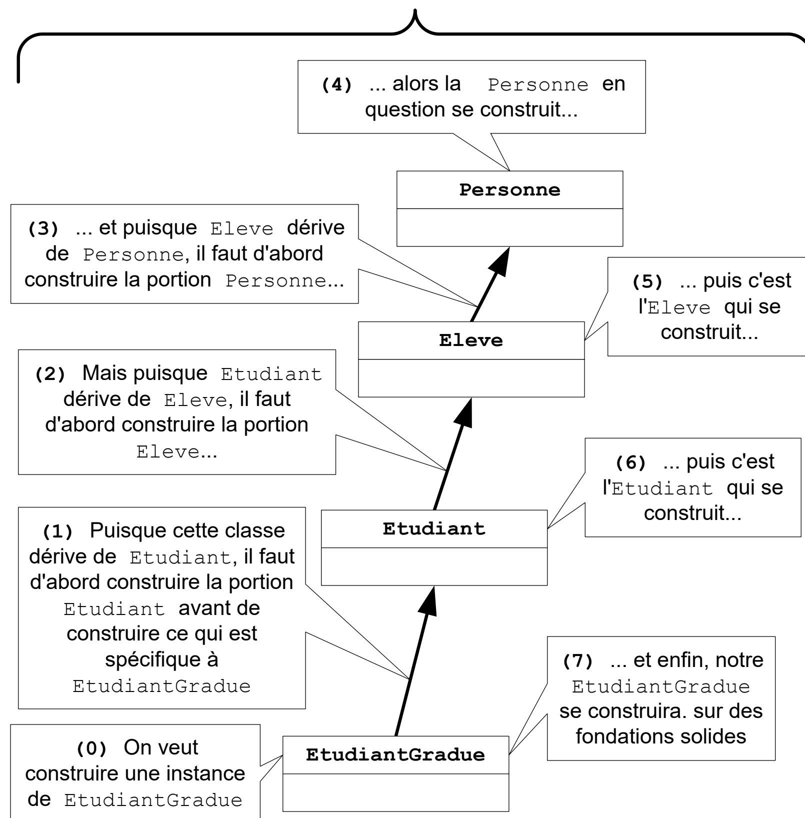
La réponse, bien sûr, est ***oui***. Dans une situation d'héritage simple comme celle examinée ici, la construction d'une instance d'une classe dérivée demande qu'on construise d'abord la partie d'elle qui appartient à sa classe parent, puis la partie qui lui est propre.

> Cela tombe sous le sens : par analogie avec la construction d'un bâtiment, on voudra d'abord construire les fondations, la classe parent, avant de se préoccuper de la construction des étages supérieurs, la classe enfant.

Prenons un exemple où on veut créer un `EtudiantGradue`, qui est une spécialisation de la classe `Etudiant`, cette dernière étant elle-même une spécialisation d'`Eleve` (*qui étudie*), cette dernière étant une spécialisation de `Personne`.

Remarquez que le processus de construction débute au bas de la hiérarchie, là où la spécialisation est la plus grande pour l'instance à construire. Cette classe demande à son parent de se construire, acte répété d'enfant en parent jusqu'à ce que la classe la plus générale, en haut de la hiérarchie, reçoive sa demande de construction.

La classe la plus générale (ici : la classe `Personne`) est alors construite. Une fois sa construction complétée, le volet `Eleve`, dérivant directement de `Personne`, sera construit, et ainsi de suite jusqu'à ce que l'instance d'`EtudiantGradue` soit construite en entier.



Prenons un autre exemple concret. Pour que celui-ci soit simple, nous allons utiliser la classe `Client`, qui est moins complexe que la classe `Eleve`.

Si, en langage C++, la classe `Personne` expose les constructeurs proposés à droite…

```
// ...
class Personne {
public:
   Personne();
   Personne(
      const string &nom, const string &prenom,
      Genre
   );
   Personne(const Personne&);
   // ...
};
```

…et si, toujours en C++, on souhaite exposer l'ensemble de constructeurs présenté à droite pour la classe `Client`… alors voici (ci-dessous) comment on pourrait écrire la définition du constructeur par défaut de `Client`, tenant compte du fait que `Client` dérive de `Personne`.

Vous remarquerez le sens à donner au constructeur de `Client` : *un `Client` par défaut est, à la base, une `Personne` typique sans dettes et dont l'adresse est inconnue.*

```
// ...
class Client
   : public Personne
{
   float montantEnSouffrance_;
   string adresse_;
public:
   Client();
   Client(
      const string &nom, const string &prenom,
      Genre, const string &adresse,
      float montantEnSouffrance
   );
   Client(const Client&);
   // ...
};
```

La préconstruction est une syntaxe naturelle pour invoquer le constructeur du parent puisque celui-ci doit être construit avant que son enfant ne le soit.

> Si nous n'avions pas explicitement invoqué un constructeur du parent à la préconstruction, son constructeur par défaut aurait été invoqué automatiquement.

```
// ...
Client::Client()
   : Personne{} // construire le parent
{
   SetMontantEnSouffrance(0.0f);
   SetAdresse("");
}
```

Une autre écriture possible serait celle à droite, qui repose strictement sur la préconstruction.

```
// ...
Client::Client()
   : Personne{},
     montantEnSouffrance_{},
     adresse_{}
{
}
```

Sachant que le constructeur par défaut est nécessairement invoqué pour le parent et les attributs si aucune indication contraire n'est donnée, on aurait pu tout simplement écrire le code plus bas et obtenir le même résultat.

Il est tout de même souhaitable de faire en sorte que l'intention soit claire et explicite dans les programmes, alors mieux vaut pêcher par excès de clarté que par excès de concision.

```
// ...
Client::Client()
   : montantEnSouffrance_{}
{
}
```

> Règle générale, il est sage de privilégier la préconstruction.

Syntaxiquement, donc, suivre l'en-tête du constructeur par : suivi d'un appel explicite au constructeur du parent suffit pour invoquer ce constructeur. Ce faisant, l'enfant peut invoquer le constructeur de son choix parmi ceux exposés par le parent. Remarquez que la construction du parent se fait avant même la première instruction du constructeur de l'enfant, incluant l'initialisation des attributs par préconstruction[18].

Note :	ce détail est important, car il est porteur de conséquences. Nous y reviendrons lorsque nous discuterons de polymorphisme.

Prenant maintenant l'exemple des constructeurs paramétriques, voici comment on procéderait, toujours pour la classe `Client` (la préconstruction des attributs est possible ici aussi, mais prudence face aux paramètres susceptibles de corrompre l'instance de `Client` en cours de construction, s'il y a lieu; les techniques proposées dans [POOv00] peuvent être précieuses ici) :

```
// exemple d'appel au constructeur paramétrique du parent
Client::Client(
   const string &nom, const string &prenom, Genre sexe,
   const string &adresse, float montantEnSouffrance
   ) : Personne{ nom, prenom, sexe }
{
   // une fois le parent construit, on enchaîne avec les trucs spécifiques à l'enfant
   SetMontantEnSouffrance(montantEnSouffrance);
   SetAdresse(adresse);
}
```

On voit que, parmi les paramètres passés à l'enfant, ceux qui servent à la construction du parent lui sont relayés par un appel à son constructeur, avant même que l'enfant ne soit construit.

Note quant à l'appel du constructeur d'un parent

Rien n'oblige le constructeur *par défaut* d'un enfant à employer le constructeur *par défaut* du parent (ceci s'applique d'ailleurs à tous les constructeurs). On pourrait par exemple, à partir du constructeur paramétrique d'un enfant, avoir recours au constructeur par défaut du parent.

En général, toutefois, on aura recours aux outils en apparence les plus appropriés. Après tout, si un constructeur du parent permet la construction par copie, pourquoi ne pas s'en servir comme base pour le constructeur par copie de son descendant? C'est le seul qui garantisse que les *parties parent* soient correctement copiées l'une dans l'autre en totalité, puisque *c'est le seul qui soit précisément fait pour ça…*

[18] Ceci est analogue à la manière avec laquelle on initialise une constante d'instance, dont la valeur doit être fixée avant que l'instance n'existe vraiment.

On s'en doute, la mécanique s'applique aussi aux constructeurs par copie (bien qu'ici, `Client` et `Personne` étant des types valeurs, on pourrait laisser s'appliquer la Sainte-Trinité et ne pas écrire cette opération du tout – ce serait même plus efficace ainsi!).

Ici, nous appliquons sans gêne la préconstruction des attributs puisque l'original est présumé valide (dû au respect du principe d'encapsulation).

```
// appel au constructeur par copie du parent
Client::Client(const Client &original)
   : Personne{original},
     montantEnSouffrance_{
        original.GetMontantEnSouffrance()
     },
     adresse_{original.GetAdresse()}
{
}
```

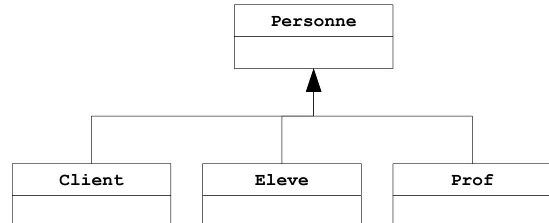
La subtilité ici tient du fait que l'*on passe en paramètre à la construction d'une* `Personne` *une instance de* `Client` (le paramètre nommé `original`). Est-ce bien là une manœuvre légale, à votre avis?

Quelques règles d'identité

Non seulement c'est légal, mais c'est très bien comme ça. Si `Client` est un descendant public de `Personne`, alors tout `Client` est aussi, au vu et au su de tous, une `Personne`. Par conséquent, on peut toujours traiter un `Client` comme une `Personne` en toute sécurité et en toute légalité[19].

Reprenant notre exemple, où :

- la classe `Eleve` dérive de `Personne`;
- la classe `Prof` dérive de `Personne`; et
- la classe `Client` dérive de `Personne`;



on notera que :

- tout `Eleve` est une `Personne`;
- tout `Prof` est une `Personne`;
- tout `Client` est une `Personne`; mais
- toute `Personne` n'est pas *nécessairement* un `Eleve` (elle pourrait aussi être un `Prof`, ou encore un `Client`, ou même autre chose n'apparaissant pas dans notre schéma).

> En `POO`, un enfant connaît ses parents mais un parent ne sait jamais combien il a véritablement d'enfants.

La règle est la suivante :

- ***tout enfant est, par définition, un cas particulier de ses parents***, et de chacun de ces ancêtres. Il offre *au moins* les mêmes opérations que ses parents. Ainsi, tout enfant peut être traité comme s'il était du même type que ses parents, sans discrimination; par contre
- ***un parent peut avoir un nombre arbitraire d'enfants***, et cela s'applique à plus forte partie à toute sa descendance. Chacun de ses enfants risque fort d'avoir des membres que le parent lui-même n'a pas. Ainsi, à moins d'être convaincu de la justesse d'une telle démarche[20], on ne peut traiter aveuglément un parent comme s'il était l'un de ses enfants.

> Il y a un bémol à ces remarques pour Java et les langages `.NET`. Voir ***Dans d'autres langages***, plus loin, pour des détails.

[19] Dans les cas d'héritage simple, du moins. En situation d'héritage multiple, on le verra, la situation demande parfois plus de prudence.

[20] …et d'utiliser, parfois, certaines opérations de conversion explicite de type un peu particulières, que nous couvrirons plus tard…

Héritage et destructeurs

Nous y reviendrons, bien sûr, mais permettons-nous une brève remarque quant au lien entre héritage et destructeurs.

S'il faut, dans un constructeur, construire le parent avant de construire l'enfant, et s'il se trouve qu'on y appelle explicitement le constructeur d'un parent avec les paramètres appropriés, la situation avec les destructeurs sera un peu différente :

- chaque classe possède un et un seul destructeur;
- celui-ci est appelé de manière déterministe par le moteur de C++ lorsqu'un objet alloué de manière automatique ou statique arrive à la fin de sa portée, de manière naturelle ou suite à la levée d'une exception, et peut aussi être appelé de manière explicite par le code client lorsque l'objet a été allouée dynamiquement;
- lorsqu'on détruit un enfant, on aimerait que son parent (et tout son arbre généalogique) soit aussi détruit;
- il doit bien y avoir un ordre dans lequel il serait préférable de détruire le parent et l'enfant;
- le seul ordre de destruction raisonnable est l'inverse de l'ordre de construction, soit les enfants avant les parents.

Nous reviendrons sur ce sujet un peu plus loin, lorsque nos objets seront devenus suffisamment complexes pour que les exemples soient pertinents.

Le mot clé ***`virtual`***, souvent apposé devant un destructeur (et devant certaines autres méthodes), jouera un rôle prédominant lorsqu'on discutera de ***polymorphisme***. Une fois ce mot clé introduit en bonne et due forme, nous serons mieux en mesure d'explorer la mécanique derrière la bonne destruction des objets susceptibles de servir en tant qu'abstraction pour une descendance potentiellement riche et complexe.

Héritage et affectation

En pratique, copier un enfant impliquer d'abord copier ses parents, ce qui doit être fait explicitement. Par exemple, si A dérive de B, alors l'opérateur d'affectation de A pourra s'écrire comme suit :

```
A& A::operator=(const A &autre) {
   B::operator=(autre); // affectation telle qu'implémentée pour le parent B
   // code propre à l'affectation pour A, peu importe ce que c'est
   return *this;
}
```

Si vous mettez en pratique l'idiome d'affectation sécuritaire [POOv00], alors la copie de la partie parent sera implicitement prise en charge par le constructeur de copie et, de manière homogène, le même opérateur d'affectation s'écrira ainsi :

```
A& A::operator=(const A &autre) {
   // la copie de autre dans un A temporaire copie aussi les parents de autre
   A{autre}.swap(*this);
   return *this;
}
```

Notez cependant que, dans bien des cas, les types valeurs pour lesquels on déploiera une gamme d'opérateurs ne sont pas les types pour lesquels l'héritage est une façon de faire appropriée. Ces classes sont habituellement instanciées pour et par elles-mêmes, pas en tant que raffinements de classes plus générales.

Nous prendrons soin, plus loin, de réfléchir aux cas pour lesquels, techniquement, l'héritage est une approche intéressante et d'identifier les cas qui s'y prêtent moins. Il est encore tôt à ce stade-ci pour en arriver à trancher de manière convaincante.

En résumé

On a vu que, dans une situation où des relations structurelles existent entre des classes, on peut approcher le problème de la représentation de ces relations d'au moins trois manières :

- développer ces classes *de manière indépendante*, faisant fi de ces relations. Ceci peut être une option viable si la relation est trompeuse (si elle n'est qu'apparente), un peu comme on n'utiliserait pas le même nom de constante pour plusieurs concepts foncièrement distincts simplement parce qu'on peut leur apposer la même valeur[21], ou si on préfère dissocier les classes dans le contexte où celles-ci sont développées;
- développer certaines d'entre elles *par composition* ou par *agrégation* d'instances d'autres classes. Ceci est une option intéressante lorsque l'une des classes peut être effectivement perçue comme un assemblage des autres (comme dans le cas de la relation qui existe entre une automobile et ses roues, par exemple) ou lorsqu'on peut définir la relation de manière possessive, à l'aide du verbe avoir;
- y aller *par héritage* est une option intéressante lorsque l'une des classes est un cas particulier de l'autre[22]. Profiter de ce type de généralisation (lorsque cela s'avère approprié, bien sûr) nous amènera à utiliser le puissant mécanisme qu'est le *polymorphisme*, que nous examinerons sous peu.

Il y a beaucoup plus à dire sur l'héritage et les bonnes pratiques `OO`. Nous y reviendrons dans la section ***Héritage et design OO*** de [POOv03].

[21] Par exemple, une constante littérale comme `0.04` pourrait être apposée à la fois à un taux annuel d'augmentation salariale et à la taille en mètres d'un objet de la vie courante. On n'utilisera pas un seul nom pour les deux idées parce qu'elles ont, pour un moment ou dans un cas particulier, la même valeur.

[22] Comme dans le cas où on traite un `Eleve` comme une `Personne`.

Empêcher la dérivation

Les trois langages que sont Java, C# et VB.NET ont ceci en commun qu'ils offrent des mécanismes formels pour empêcher la dérivation d'une classe donnée; pour des raisons philosophiques, C++ fait de même mais seulement depuis C++ 11.

Pour des raisons qui deviendront plus claires quand nous aurons abordé le polymorphisme, ce mécanisme est particulièrement important en Java.

En Java, une classe qualifiée `final` (ne peut servir de parent à d'autres classes.

```
public final class X {
   // ...
}
```

En C#, on arrive au même effet en utilisant une classe qualifiée `sealed`.

```
public sealed class X
{
   // ...
}
```

Pour sa part, le langage VB.NET utilise plutôt le mot clé `NotInheritable` pour obtenir un effet semblable.

```
Public NotInheritable Class X
End Class
```

En C++, à l'image de Java, le mot clé `final` est utilisé pour marquer qu'une classe n'aura pas d'enfant, mais la position de ce mot clé n'est pas la même dans les deux langages.

```
class X final {
   // ...
};
```

Nous reviendrons au moment opportun sur la question du bon usage de ces mots clés, à savoir pourquoi certaines classes pourraient vouloir s'assurer de ne jamais avoir d'enfants. Cette réflexion sera amorcée dans la section ***Classes terminales***, plus bas.

Dans d'autres langages

Les langages Java, C# et VB.NET supportent moins de variantes et de nuances que C++ pour ce qui est de l'héritage, ce qui deviendra de plus en plus clair alors que nous progresserons dans notre étude de ce concept.

Ce que nous observerons ici sera l'implémentation, dans ces trois langages, de l'héritage dans son sens le plus simple, qu'on nomme aussi héritage d'implémentation, et qui correspond à ce qui a été couvert plus haut avec les classes Personne, Client, Eleve et Prof. Dans chacun des langages explorés à l'intérieur de la présente section, nous examinerons le cas d'une classe Client très simple et dérivée d'une classe Personne décrite dans ce langage. Rien ne vous empêche, si le cœur vous en dit, d'implémenter par vous-mêmes les autres classes de l'exemple.

En Java, la classe Personne se présente comme suit (à droite).

Les constructeurs vont de soi. Notez que ce qui joue le rôle de constructeur de copie (protégé, mais nous y reviendrons en temps et lieu) lève une exception lorsque le paramètre qui lui est supplée s'avère nul.

Java, comme C# et VB.NET, ne permet de manipuler des objets que de manière indirecte, par des références, ce qui demande une prudence plus grande dans cette méthode qu'en C++, où le paramètre existe nécessairement au préalable.

Les accesseurs sont banals, du fait que les types impliqués sont immuables.

```
public class Personne {
   public enum Genre {
      Féminin, Masculin
   };
   public Personne() {
      this ("", "", Genre.Féminin);
   }
   public Personne(String nom, String prénom, Genre g) {
      setNom(nom);
      setPrénom(prénom);
      setSexe(g);
   }
   protected Personne(Personne p)
      throws NullPointerException {
      if (p == null) {
         throw new NullPointerException();
      }
      setNom(p.getNom());
      setPrénom(p.getPrénom());
      setSexe(p.getSexe());
   }
   public String getNom() {
      return nom;
   }
   public String getPrénom() {
      return prénom;
   }
   public Genre getSexe() {
      return sexe;
   }
```

Notez que j'ai choisi d'utiliser l'opérateur ternaire `?:` dans les mutateurs par souci de concision, mais qu'il aurait été tout à fait valable d'y aller, dans chaque cas, d'une alternative classique.

La méthode protégée (oui, nous allons incessamment discuter de ce mot) `copies()` réalise une copie de valeur, à la manière de l'opérateur d'affectation en C++. Elle suit la norme appliquée pour ce qui tient du constructeur de copie (plus haut) en levant une exception lorsque son paramètre est nul.

La méthode `equals()` réalise une comparaison de contenu, à l'image de l'opérateur `==` de C++. Notez que, pour tout objet `x`, la règle en Java est que `x.equals(null)` doit retourner `false`.

> Notez au passage qu'il est très malsain de comparer deux nombres à virgule flottante avec `==` comme nous le faisons ici.

```
   public void setNom(String nom) {
      this.nom = (nom != null)? nom : "";
   }
   public void setPrénom(String prénom) {
      this.prénom = (prénom != null)? prénom : "";
   }
   public void setSexe(Genre g) {
      sexe = g;
   }
   protected void copies(Personne p)
      throws NullPointerException {
      if (p == null) {
         throw new NullPointerException();
      }
      setNom(p.getNom());
      setPrénom(p.getPrénom());
      setSexe(p.getSexe());
   }
   public boolean equals(Object o) {
      Personne p = (Personne) o;
      return p != null &&
             getNom().equals(p.getNom()) &&
             getPrénom().equals(p.getPrénom()) &&
             getSexe() == p.getSexe();
   }
   private String prénom, nom;
   private Genre sexe;
}
```

Toujours en Java, l'héritage s'exprime par le mot clé **`extends`**. On dit que `D` dérive de `B` si `D` *étend* `B` (donc si `class D extends B`). Pour exprimer le fait que `Client` soit une spécialisation de `Personne`, on déclarera donc (sans surprise) que `Client extends Personne`.

En Java, une classe ne peut avoir qu'un seul parent. Pour cette raison, toute référence faite au parent d'une classe se fait par le mot clé **`super`**. En particulier, appeler le constructeur du parent se fait en utilisant **`super()`** comme une méthode avec les paramètres appropriés. Un appel à `super()` doit d'ailleurs être la toute première opération à apparaître dans un constructeur.

> On ne peut utiliser à la fois `super()` et `this()` dans un même constructeur (ce qui, en y pensant bien, est raisonnable).

Les accesseurs et les mutateurs proposés à droite sont quelque peu naïfs. En pratique, si vous souhaitez les implémenter, il y aurait lieu de les raffiner, par exemple en levant des exceptions quand les paramètres reçus par ces méthodes divergent des politiques de validité en place.

En particulier, `setAdresse()` gagnerait à lever une exception dans le cas où son paramètre est nul plutôt que de masquer le problème comme elle le fait ici

```
class Client extends Personne {
   public Client() {
      super();
      setAdresse("");
      setDette(0.0f);
   }
   public Client
      (String Nom, String Prénom,
       Genre Sexe, String Adresse, float Dette) {
      super(Nom, Prénom, Sexe);
      setAdresse(Adresse);
      setDette(Dette);
   }
   protected Client(Client c) {
      super(c);
      setAdresse(c.getAdresse());
      setDette(c.getDette());
   }



   public String getAdresse() {
      return adresse;
   }
   public float getDette() {
      return dette;
   }
   public void setAdresse(String adresse) {
      if (adresse != null) {
         this.adresse = adresse;
      }
   }
   public void setDette(float dette) {
      this.dette = dette;
   }
```

Remarquez les méthodes `copies()` et `equals()` qui délèguent une partie de leur tâche au parent, pour respecter le principe d'encapsulation, et qui ne traitent ce qui leur est propre qu'une fois le travail de fondation terminé.

Remarquez que le type public `Genre` du parent `Personne` est hérité par l'enfant `Client` et lui appartient tout autant qu'il appartient à son parent.

```
   protected void copies(Client c) {
      if (c != null) {
         super.copies(c);
         setAdresse(c.getAdresse());
         setDette(c.getDette());
      }
   }
   public boolean equals(Object o) {
      Client c = (Client) o;
      return super.equals(c) &&
             getAdresse().equals(c.getAdresse()) &&
             getDette() == c.getDette();
   }
   private String adresse;
   private float dette;
}
```

En C#, la classe `Personne` se déclinerait à peu près tel que suggéré par l'exemple à droite.

Sans grande surprise, la notation ressemble considérablement à celle de Java. Par souci de conformité, je n'ai pas inséré les déclarations d'opérateurs (qui ne nous intéressent pas vraiment ici de toute manière).

Les remarques apposées au code Java s'appliquent ici aussi.

```
public class Personne
{
   public enum Genre
   {
      Féminin, Masculin
   };
   public Personne()
      : this("", "", Genre.Féminin)
   {
   }
   public Personne
      (string nom, string prénom, Genre g)
   {
      Nom = nom;
      Prénom = prénom;
      Sexe = g;
   }
   protected Personne(Personne p)
   {
      if (p == null)
         throw new NullReferenceException();
      Nom = p.Nom;
      Prénom = p.Prénom;
      Sexe = p.Sexe;
   }
```

```
   public string Nom
   {
      get { return nom; }
      set
      {
         nom = (value != null)? value : "";
      }
   }
   public string Prénom
   {
      get { return prénom; }
      set
      {
         prénom = (value != null)? value : "";
      }
   }
   public Genre Sexe
   {
      get; set;
   }
   protected void Copies(Personne p)
   {
      if (p == null)
         throw new NullReferenceException();
      Nom = p.Nom;
      Prénom = p.Prénom;
      Sexe = p.Sexe;
   }
   public override int GetHashCode()
   {
      return base.GetHashCode(); // bof
   }
   public override bool Equals(Object o)
   {
      Personne p = (Personne) o;
      return p != null &&
             Nom.Equals(p.Nom) &&
             Prénom.Equals(p.Prénom) &&
             Sexe == p.Sexe;
   }
   private string prénom, nom;
}
```

Tel que mentionné dans [POOv00], C# a ce vilain défaut d'obliger la rédaction de `GetHashCode()` pour toute classe définissant `Equals()`, alors que la relation entre ces deux idées n'est pas biunivoque (l'une implique l'autre mais l'inverse est faux). L'implémentation de `GetHashCode()` utilisée ici est (très) déficiente mais correspond à celle proposée par *Visual Studio*.

Nous reviendrons sur le mot clé `override` un peu plus loin, quand nous discuterons du polymorphisme.

La classe `Client` en C# ressemblera quant à elle à l'exemple proposé à droite.

Le mot clé super de Java a son équivalent en C# avec le mot clé **`base`**, qui sert dans les mêmes circonstances (à la fois en tant qu'attribut et en tant que méthode) et pour les mêmes raisons.

Remarquez par contre une nuance dans la manière qu'a un constructeur de classe dérivée de solliciter le constructeur de son parent.

Ici, C# ressemble un peu plus à C++ en nous demandant de placer l'appel à `base()` avant l'accolade ouvrante du constructeur de la classe dérivée.

Pour le reste, les remarques apposées au code Java, plus haut, s'appliquent intégralement ici aussi, dans la mesure où `base` et utilisé en C# et `super` est utilisé en Java.

```
public class Client : Personne
{
   public Client() : base()
   {
      Adresse = "";
      Dette = 0.0f;
   }
   public Client
      (string nom, string prénom, Genre sexe,
       string adresse, float dette)
      : base(nom, prénom, sexe)
   {
      Adresse = adresse;
      Dette = dette;
   }
   protected Client(Client c)
      : base(c)
   {
      Adresse = c.Adresse;
      Dette = c.Dette;
   }
   public string Adresse
   {
      get { return adresse; }
      set
      {
         if (value != null)
            adresse = value;
      }
   }
   public float Dette
   {
      get; set;
   }
```

```
   protected void Copies(Client c)
   {
      if (c == null)
         throw new NullReferenceException();
      base.Copies(c);
      Adresse = c.Adresse;
      Dette = c.Dette;
   }
   public override int GetHashCode()
   {
      return base.GetHashCode(); // bof
   }
   public override bool Equals(Object o)
   {
      Client c = (Client) o;
      return base.Equals(c) &&
             Adresse.Equals(c.Adresse) &&
             Dette == c.Dette;
   }
   private string adresse;
}
```

Notez au passage qu'il est très malsain de comparer deux nombres à virgule flottante avec == comme nous le faisons ici.

Enfin, en VB.NET, la classe Personne se déclinerait à peu près tel que suggéré par l'exemple à droite.

Il s'agit pour l'essentiel de code évident compte tenu de ce que nous avons exploré jusqu'ici au sujet de ce langage.

```
Public Class Personne
   Public Enum Genre
      Féminin
      Masculin
   End Enum
   Public Sub New()
      Me.New("", "", Genre.Féminin)
   End Sub
   Public Sub New(ByVal Nom As String, _
                  ByVal Prénom As String, _
                  ByVal g As Genre)
      SetNom(Nom)
      SetPrénom(Prénom)
      SetSexe(g)
   End Sub
   Public Sub New(ByVal p As Personne)
      If (p Is Nothing) Then
         Throw new NullReferenceException
      End If
      SetNom(p.GetNom())
      SetPrénom(p.GetPrénom())
      SetSexe(p.GetSexe())
   End Sub
```

Le mot clé `Overloads` ici joue le même rôle que celui joué par le mot clé `override` de C#.

Nous reviendrons sur chacun dans la section portant sur le polymorphisme, plus loin.

```
   Public Function GetNom() As String
      GetNom = nom_
   End Function
   Public Function GetPrénom() As String
      GetPrénom = prénom_
   End Function
   Public Function GetSexe() As Genre
      GetSexe = sexe_
   End Function
   Public Sub SetNom(ByVal nom As String)
      If (Not nom Is Nothing) Then
         nom_ = nom
      End If
   End Sub
   Public Sub SetPrénom(ByVal prénom As String)
      If (Not prénom Is Nothing) Then
         prénom_ = prénom
      End If
    End Sub
   Public Sub SetSexe(ByVal g As Genre)
      sexe_ = g
   End Sub
   Public Sub Copies(ByVal p As Personne)
      If (p Is Nothing) Then
         Throw new NullReferenceException
      End If
      SetNom(p.GetNom())
      SetPrénom(p.GetPrénom())
      SetSexe(p.GetSexe())
   End Sub
   Public Overloads Function GetHashcode() As Integer
      Return MyBase.GetHashCode
   End Function
   Public Overloads Function Equals(ByVal o As Object) _
         As Boolean
      Dim p As Personne = CType(o, Personne)
      Return Not p Is Nothing And _
            GetNom().Equals(p.GetNom()) And _
            GetPrénom().Equals(p.GetPrénom()) And _
            GetSexe() = p.GetSexe()
   End Function
   Private prénom_ As String
   Private nom_ As String
   Private sexe_ As Genre
End Class
```

En VB.NET, l'héritage s'exprime par le mot clé Inherits, qui doit être une instruction suivant la déclaration de la classe sans toutefois pouvoir se situer sur la même ligne que cette déclaration.

Le constructeur d'un enfant invoque le constructeur de son parent en appelant MyBase.New(); de manière générale, un enfant appelle une méthode *M*() de son parent en invoquant MyBase.*M*().

Pour le reste, les remarques qui étaient applicables dans le cas de Java et (surtout) dans le cas de C#, plus haut, demeurent valides ici aussi.

```
Public Class Client
   Inherits Personne
   Public Sub New()
      MyBase.New()
      SetAdresse("")
      SetDette(0.0F)
   End Sub
   Public Sub New(ByVal nom As String, _
                  ByVal prénom As String, _
                  ByVal sexe As Genre, _
                  ByVal adresse As String, _
                  ByVal dette As Single)
      MyBase.New(nom, prénom, sexe)
      SetAdresse(adresse)
      SetDette(dette)
   End Sub
   Public Sub New(ByVal c As Client)
      MyBase.New(c)
      SetAdresse(c.GetAdresse())
      SetDette(c.GetDette())
   End Sub
   Public Function GetAdresse() As String
      GetAdresse = adresse_
   End Function
   Public Function GetDette() As Single
      GetDette = dette_
   End Function
   Public Sub SetAdresse(ByVal adresse As String)
      If (Not adresse Is Nothing) Then
         adresse_ = adresse
      End If
   End Sub
   Public Sub SetDette(ByVal dette As Single)
      dette_ = dette
   End Sub
```

```
   Public Overloads Sub Copies(ByVal c As Client)
      If (Not c Is Nothing) Then
         MyBase.Copies(c)
         SetAdresse(c.GetAdresse())
         SetDette(c.GetDette())
      End If
   End Sub
   Public Overloads Function GetHashcode() As Integer
      Return MyBase.GetHashCode
   End Function
   Public Overloads Function Equals(ByVal o As Object) _
         As Boolean
      Dim c As Client = CType(o, Client)
      Equals = MyBase.Equals(c) And _
               GetAdresse().Equals(c.GetAdresse()) And _
               GetDette() = c.GetDette()
   End Function
   Private adresse_ As String
   Private dette_ As Single
End Class
```

Exercices – Série 00

EX00 – Pour vous pratiquer avec la syntaxe de l'héritage et des constructeurs, essayez d'implanter en entier la classe Personne et les classes Eleve, Prof et Client.

EX01 – Implantez la classe Carre de manière à ce qu'elle soit un enfant de la classe Rectangle. Est-ce que ceci simplifie la programmation de la classe Carre? *Portez une attention spéciale aux mutateurs de Carre* : y a-t-il quelque chose auquel il faut porter une attention particulière? Peut-on, dans la classe Carre, utiliser implicitement et tels quels les mutateurs de sa classe parent?

Implémenter Carre comme un enfant de Rectangle est un exemple connu de cas pathologique qui met en relief des particularités du bon (et du mauvais) usage de l'héritage. Nous y reviendrons sous peu.

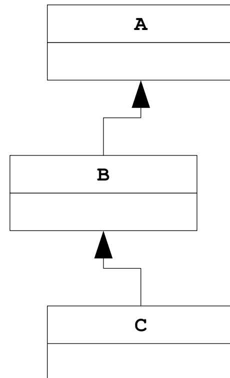
EX02 – Peut-on imaginer un ancêtre commun, Forme, aux classes Rectangle et Triangle développées plus tôt dans la session? Aurait-elle des attributs et des méthodes qui lui seraient propres? Lesquels?

EX03 – *Cet exercice est un peu plus volumineux*. Réalisez le travail proposé à ***Exercices – Série 10)*** de [POOv00]. Pourrait-on modifier la classe Joueur de ce travail pour qu'elle devienne un enfant de la classe Personne, sans que cela n'ait la moindre influence sur le programme principal qui utilisait des instances de Joueur? Si oui, expliquez ce qu'il faudrait faire pour y arriver. Sinon, expliquez pourquoi.

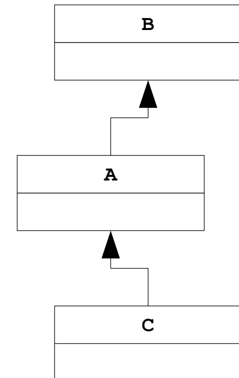
EX04 – En début de section, nous avons décidé d'utiliser un type énuméré, Genre, pour représenter le sexe d'une Personne. Il pourrait être intéressant de définir Genre comme étant une classe. Pouvez-vous imaginer ce à quoi une telle classe pourrait ressembler? Quel impact cette décision aurait-elle sur le code client? L'héritage entre-t-il dans votre conception de la classe Genre?

Réflexions

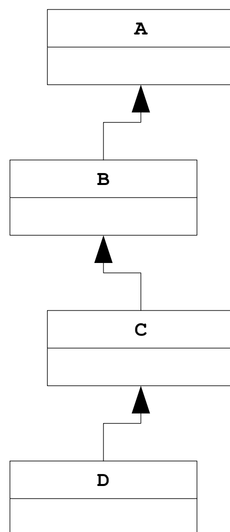
Est-il possible d'envisager, pour les classes `A`, `B` et `C`, la relation d'héritage décrite par le schéma `UML` proposé à droite? Expliquez votre réponse.



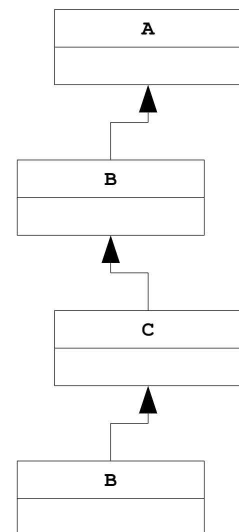
Est-il possible d'envisager, pour les classes `A`, `B` et `C`, la relation d'héritage décrite par le schéma `UML` proposé à droite? Expliquez votre réponse.



Est-il possible d'envisager, pour les classes `A`, `B`, `C` et `D`, la relation d'héritage décrite par le schéma `UML` proposé à droite? Expliquez votre réponse.



Est-il possible d'envisager, pour les classes `A`, `B` et `C`, la relation d'héritage décrite par le schéma `UML` proposé à droite? Expliquez votre réponse.



Catégories d'héritage

Pourquoi cette section?

Le langage C++ donne une granularité plutôt fine à l'idée d'héritage. Nous examinerons ici les nuances des différentes spécifications d'accès à notre disposition. Notez que cette section touche à des points que vos spécialistes techniques peuvent ne pas maîtriser en détail; ils relèvent de réflexions de plus haut niveau quant au design et à la sécurité. C'est d'ailleurs pourquoi *nous* nous y intéresserons.

Maintenant que nous avons examiné ce qu'est, à la base, l'héritage, nous allons examiner de plus près l'impact global que comporte la décision de dériver une classe d'une autre classe, et les manières d'y arriver.

Nous maintiendrons pour l'instant notre approche se concentrant sur l'héritage simple, donc le type d'héritage obtenu lorsque chaque classe enfant n'a qu'un et un seul parent, et nous nous demanderons *ce qui*, dans une situation d'héritage, est *effectivement* hérité.

Les mots clés comme `private` et `public` auront un impact sur notre étude, et nous en profiterons pour introduire un autre mot clé, `protected`, qui ne prend son sens que dans un contexte d'héritage.

Spécifications d'accès et héritage

On retrouve trois spécifications d'accès pour l'héritage simple en C++. À chacune correspond un mot clé parmi les mots `public`, `private` et `protected`. Ce qui suit se veut un examen détaillé de chacune de ces spécifications : sa raison d'être, ses implications, ses champs d'application, ses atouts et ses inconvénients[23].

Bien comprendre ces spécifications et leur rôle dans une hiérarchie d'objets aide à poser un jugement éclairé sur un design objet, sans vraiment avoir à savoir comment le code des différentes méthodes des classes impliquées sera implanté.

Il est d'ailleurs probable que vous ayez à guider des gens plus versés que vous techniquement, mais qui seront moins au fait que vous de ces idées à portée plus vaste qui feront le corps de ce document. La mise en commun de votre vision et de leur technique formera, souhaitons-le, une combinaison gagnante.

Dans les exemples qui suivent, nous utiliserons parfois des noms génériques de classe pour expliquer les concepts reliés à l'héritage : la classe parente s'appellera souvent `Parent` ou simplement `B` (pour *base*) et la classe enfant s'appellera fréquemment `Enfant`, ou `D` (pour *dérivée*). Au besoin, transposez ceci dans une relation concrète qui vous conviendra mieux (prenez `Forme` pour parent et `Rectangle` pour enfant, par exemple).

[23] Ce qui sera discuté ici n'a pas de réel équivalent en Java ou en C#. Comme à l'habitude, lorsqu'un langage ne supporte pas lui-même une idée, on doit s'en remettre (si on souhaite tout de même la supporter) à la discipline des membres de l'équipe de développement.

Héritage et consommation d'espace

En C++, comme en C, l'opérateur statique `sizeof()` permet d'obtenir la taille, en *bytes*, d'un objet (le terme « objet » étant pris au sens très large d'une entité d'un certain type). Par définition, `sizeof(char)==1`[24], mais la taille de la plupart des autres types n'est pas pleinement fixée par la norme du langage (contrairement à ce qui prévaut en Java ou dans les langages `.NET` par exemple).

Tout objet C++ occupe, pour des raisons techniques (demandez à votre professeur de vous l'expliquer en personne), un espace non nul, qui est supérieur ou égal à la somme des tailles de ses membres. Ainsi, dans le programme suivant affichera `"vrai!"` et `"oui!"` :

```
#include <string>
#include <iostream>
// ...using...
class X {
   char c_;
   int i_;
   string s_;
   float *pf_;
   // ...
};
class Y {};
int main() {
   if(sizeof(X) >= sizeof(char)+sizeof(int)+sizeof(string)+sizeof(float*))
      cout << "vrai!" << endl;
   else
      cout << "faux?" << endl;
   if (sizeof(Y) > 0) // tautologie
      cout << "oui!" << endl;
   else
      cout << "non?" << endl;
}
```

Bien que nous ayons ici utilisé `sizeof(X)` et `sizeof(Y)`, nous aurions pu appliquer l'opérateur `sizeof()` sur une instance de l'une ou l'autre de ces classes, et nous aurions eu le même résultat.

Sans grandes surprises, l'héritage a un impact sur la taille des enfants : la taille d'un objet est au moins la somme des tailles de ses attributs, à laquelle s'ajoute la taille de son parent (ou, dans un cas d'héritage multiple, la somme des tailles de ses parents). Il y a bien sûr des nuances, dans le cas d'héritage virtuel (voir ***Héritage virtuel***) ou de polymorphisme (voir ***Polymorphisme***), mais c'est l'idée de base.

[24] Ceci signifie qu'un `char` occupe un *byte* d'espace mémoire, ce qui signifie au moins huit bits.

Optimisation EBCO

Par contre, un compilateur C++ peut procéder à une optimisation nommée ***Empty Base Class Optimization***, ou EBCO [hdEBCO]. Cette optimisation permet, quand le parent d'une classe ne possède pas d'attributs, de réduire à zéro l'espace occupé par le parent dans l'enfant. Le gain d'échelle peut être important.

Ainsi, examinez le code suivant :

```
class X {};
class Y {
   int val;
   X x;
   // ...
};
class Z : X {
   int val;
   // ...
};
```

Ici, nous savons plusieurs choses sur la taille qu'occuperont nos objets en mémoire :

- en soi, `sizeof(X)>0`, tel que mentionné plus haut;
- nous savons que `sizeof(Y)>=sizeof(int)+sizeof(X)`; mais
- il se trouve que `sizeof(Z)>=sizeof(int)`, et qu'il est très probable que `sizeof(Z)==sizeof(int)`. C'est là un impact direct d'EBCO.

L'optimisation EBCO n'est applicable que si la classe parent n'a aucun attribut d'instance; la classe parent peut toutefois avoir des méthodes d'instance et des membres de classe. L'immense majorité des compilateurs C++ appliquent EBCO systématiquement, du moins pour les cas d'héritage simple.

Héritage public

```
class Enfant
   : public Parent
{
   // ...
};
```

L'héritage public repose sur l'emploi du mot clé `public` précédant le nom du parent lors de la déclaration de la classe enfant (voir à droite).

On peut dire que l'héritage public signifie que *tout* `Enfant` *est aussi un* `Parent`*, et tout le monde le sait.*

Le sens opérationnel de cette phrase est que tout objet, tout sous-programme peut tirer profit de l'information à l'effet que tout `Enfant` est aussi un `Parent`.

Impact de cette affirmation : il est possible de passer en paramètre un `Enfant` à tout sous-programme demandant qu'on lui passe un `Parent`.

À titre d'exemple, si `Client` hérite publiquement de `Personne`, alors un sous-programme prenant en paramètre une `Personne` peut sans problème recevoir en paramètre une instance de `Client`, ceci parce qu'un `Client` est alors une `Personne` au su et au vu de tous.

Illustration : si la classe `Personne` existe, et si la classe `Client` hérite publiquement de `Personne`, alors ce qui suit est valide :

```
// Procédure contacter()
// Intrants: une Personne
// Extrants: ---
// Rôle:     contacter la personne passée en paramètre
void contacter(const Personne&);

// Procédure contacter_mauvais_payeur()
// Intrants: un Client
// Extrants: ---
// Rôle:     si le client passé en paramètre est un mauvais payeur,
//           le contactera
void contacter_mauvais_payeur(const Client &c) {
   if (c.solde() < 0) // si le client doit des sous à l'entreprise
      contacter(c); // On traite le Client comme une Personne
}
```

Note : dans l'exemple ci-dessus, `if (c.est_mauvais_payeur())` aurait été préférable à `if (c.solde() < 0)` Pouvez-vous expliquer pourquoi ?

Question piège : aurait-il été préférable de nommer la fonction `contacter_personne()` plutôt que `contacter()` ? Expliquez votre position.

Héritage public et membres publics

Sans surprises, lorsqu'un enfant hérite publiquement d'un parent, il a accès à tous les membres publics de son parent. Il en va de même pour les sous-programmes qui utilisent l'enfant.

Corollaire : un objet ou un sous-programme utilisant une instance d'`Enfant` peut utiliser, à travers celle-ci, *les membres publics* de `Parent`, puisqu'il est du domaine public qu'un `Enfant` est aussi un `Parent`.

Illustration : présumant que `Personne` dévoile la méthode *publique* `nom()` et que `Client` dérive *publiquement* de `Personne`, alors il est possible pour un sous-programme manipulant une instance de `Client` d'utiliser sa méthode `nom()` :

Personne.h

```
#include <string>
class Personne {
   // ...
public:
   std::string nom() const;
   // ...
};
```

Client.h

```
#include "Personne.h"
class Client : public Personne {
   // ...
};
```

Exemple.cpp

```
#include "Client.h"
#include <iostream>
int main() {
   using namespace std;
   Client c;
   // ...
   cout << c.nom() << endl; // méthode de Personne via un Client
   // ...
};
```

Il y a des nuances pour ce qui est des méthodes effectivement héritées d'un parent public. L'une d'elles est que les constructeurs ne sont pas hérités : l'enfant doit avoir les siens, et doit appeler volontairement les constructeurs du parent dont il a besoin. L'autre a trait à une question de contexte et de portée; voir la section ***Héritage, surcharge de méthodes et espaces nommés***, plus loin dans ce document.

Héritage public et membres privés

Même si un enfant hérite publiquement d'un parent, les membres privés du parent demeurent privés au parent. Cela signifie que l'enfant ne peut pas avoir directement accès aux membres privés de son parent, pas plus que tout autre code client d'ailleurs. Privé signifie vraiment privé.

Illustration : présumons la classe `Client` possédant la méthode privée `SetSolde()` permettant de modifier son solde dû, de même que la (clairement mal nommée) constante publique `TAUX` et les méthodes publiques `deposer()` et `retirer()` pour ajouter ou déduire du solde courant.

Si une classe `ClientPrivilege` hérite publiquement de `Client`, alors une instance de `ClientPrivilege` pourra utiliser `deposer()` et `retirer()`, mais pas `SetSolde()`. Sans surprises, l'attribut `solde_` de `Client` ne sera pas accessible à une instance de `ClientPrivilege`.

Client.h

```
#include "Personne.h"
class Client : public Personne {
   // ...
private:
   float solde_;
   void SetSolde(float);
   // ...
public:
   float solde() const noexcept;
   void deposer(float);
   void retirer(float);
   static const float TAUX;
   // ...
};
```

ClientPrivilege.h

```
#include "Client.h"
class ClientPrivilege : public Client {
   // ...
public:
   void calculer_interet();
   static const float TAUX_PRIVILEGE,
                      SEUIL_PRIVILEGE;
};
```

ClientPrivilege.cpp

```
#include "ClientPrivilege.h"
// ...
void ClientPrivilege::calculer_interet() {
   float soldeCourant = solde(),
         interet;
   if (soldeCourant > SEUIL_PRIVILEGE)
      interet = soldeCourant * TAUX_PRIVILEGE;
   else // TAUX vient de Client
      interet = soldeCourant * TAUX;
   // Incapable d'utiliser SetSolde() ou
   // solde_, qui sont privés...
   retirer(soldeCourant);
   deposer(soldeCourant + interet);
}
```

Note : certains problèmes demandent qu'on puisse garantir que les opérations de retrait et de dépôt soient toutes deux réalisées, quoiqu'il arrive. Il existe des moyens d'offrir de telles garanties, mais ces moyens relèvent plus d'un cours de bases de données ou de systèmes client/ serveur.

Héritage public et membres protégés

Avec l'héritage, une nouvelle catégorie (nouvelle *pour nous*, bien sûr) s'ajoute aux spécifications déjà connues que sont `public` et `private` : la spécification ***protégé*** (mot clé **`protected`**).

⇒ Les membres d'une classe qui sont spécifiés **`protected`** apparaissent aux descendants de cette classe comme s'ils étaient *publics*, et au monde extérieur comme s'ils étaient *privés*.

La notation appliquée dans les schémas `UML` propose qu'on précède un membre protégé du signe `#` (*dièse*).

On utilisera des membres protégés lorsqu'on souhaite les cacher des utilisateurs de notre classe et des utilisateurs de ses enfants, mais pas des enfants eux-mêmes.

Le toujours pertinent ***Pierre Prud'homme*** m'a un jour fait part de cette réflexion d'une équipe de travail dans l'un de ses cours quant à certaines qualifications apparaissant en `POO`…

Réflexions sur l'application du modèle objet au fonctionnement d'équipe :

- *membre* `public` *: qui collabore avec les autres membres de l'équipe;*
- *membre* `protected` *: qui collabore avec un ou deux privilégiés seulement;*
- *membre* `private` *: qui garde son code au fond d'une boîte noire. Il y donnera accès la veille de la remise du* `TP`*;*
- *méthode* `virtual` *: qui délègue à un autre membre la résolution d'un problème complexe;*
- *méthode abstraite : méthode impossible à comprendre pour l'instant. On s'y attaquera plus tard si le temps le permet.*

Dans l'exemple du `ClientPrivilege`, plus haut, indiquer que `SetSolde()` est protégée (plutôt que privée) dans la classe `Client` aurait permis de l'utiliser dans la méthode `calculer_interet()` de `ClientPrivilege`, nous évitant ainsi d'utiliser successivement `retirer()` et `deposer()`.

`Client.h` (modifié)

```
#include "Personne.h"
class Client : public Personne {
   float solde_;
   // ...
protected:
   void SetSolde(float);
   // ...
public:
   float solde() const noexcept;
   void deposer(float);
   void retirer(float);
   static const float TAUX;
   // ...
};
```

`ClientPrivilege.cpp` (modifié)

```
#include "ClientPrivilege.h"
const float
   ClientPrivilege::TAUX_PRIVILEGE = 0.18f,
   ClientPrivilege::SEUIL_PRIVILEGE = 1000.0f;
// ...
void ClientPrivilege::calculer_interet() {
   float soldeCourant = solde(), interet;
   if (soldeCourant > SEUIL_PRIVILEGE)
      interet = soldeCourant * TAUX_PRIVILEGE;
   else // TAUX vient de Client
      interet = soldeCourant * TAUX;
   // SetSolde() est spécifiée protected dans la
   // classe Client, donc la classe ClientPrivilege,
   //, qui en héritepubliquement, peut l'utiliser
   SetSolde(soldeCourant + interet);
}
```

Déterminer si un `SetSolde()` protégé est préférable à un autre qui serait privé ici est une question d'ordre politique. Si vous et votre équipe décidez que cet usage est à privilégier, alors il devient bon *pour vous*. La spécification `protected` est un outil, pas un dogme.

Notes philosophiques sur l'héritage public

Dans la grande majorité des cas, l'héritage public est ce que nous avons en tête lorsque nous parlons d'héritage. Il s'agit du seul type d'héritage possible dans la plupart des langages, dont Java et les langages .NET.

L'héritage public dénote une relation en fonction du verbe ***être*** : l'enfant y **est** un cas particulier du parent. La communauté OO anglophone parle d'ailleurs d'un *Is-A Relationship* dans ce cas, plutôt que d'un *Has-A Relationship* (relation sur le mode du verbe ***avoir***), qui est le type de relation exprimé par la composition ou l'agrégation[25] (et, en partie, nous le verrons sous peu, par les notions d'héritage privé et d'héritage protégé).

Il existe d'autres types de relations, par exemple une relation ***est implémenté en termes de*** (*Is Implemented In Terms Of*) qui décrit le fruit de certaines pratiques OO associées à ce qu'on nomme les interfaces (plus bas).

Lorsque nous discuterons des idées derrière l'***héritage d'interface***, nous aurons surtout recours à l'idée d'héritage public dans nos descriptions, surtout parce que c'est avec cette manière d'exprimer l'héritage qu'on parvient le mieux à représenter l'idée selon laquelle le parent est une généralisation de l'enfant, et selon laquelle il est possible de regrouper tous les enfants d'un même parent pour les traiter en fonction de leurs points en commun.

Barbara Liskov a montré que l'héritage public signifie la substituabilité [LiskovSubs]. Dans ses termes, D est un enfant public correct de B si on peut utiliser un D partout où il serait normalement possible d'utiliser un B. Il est important que le comportement d'un enfant soit sémantiquement conforme aux attentes placées sur le parent.

[25] Rappel: la **composition** a été couverte précédemment, à titre comparatif pour introduire l'idée d'héritage (au sens public) alors que l'**agrégation**, sa cousine, ne l'a pas été. Pour en savoir plus sur l'agrégation, voir ***Association, agrégation et composition***. Aussi, prudence avec le mot anglais *aggregate* qui apparaît parfois dans la documentation de C++ et qui réfère de manière élargie à toute entité non scalaire (incluant entre autres les struct et les tableaux), et qui ne veut pas indiquer la même idée.

Exercices – Série 01.0

Présumant les classes suivantes :

```
class X {
   int valeur_;
public:
   X();
   ~X();
   int valeur() const noexcept;
   void SetValeur(int);
protected:
   void SetValeurCru(int);
};
```

```
#include "X.h"
class Y : public X {
public:
   Y();
   ~Y();
   int valeur_surprise() const noexcept;
   void CalculerValeur(int, int);
};
```

Indiquez, pour chaque sous-programme ci-dessous, s'il est correctement écrit ou non (on entendra ici *correct* au sens de *n'enfreignant pas les règles du langage*), et expliquez pourquoi.

EX00

Code	Correct	Incorrect

`int X::valeur() const noexcept {`
`   return valeur_;`
`}`		

Explications : ____________________

____

____

____

EX01

Code	Correct	Incorrect

`void X::SetValeur(int val) {`
`   if (val > 0) {`
`      SetValeurCru(val);`
`   }`
`}`		

Explications : ____________________

____

____

____

		Correct	Incorrect

EX02	`void X::SetValeurCru(int val) {`
	`valeur_ = val;`
	`}`		

Explications : ____

____

____

____

		Correct	Incorrect

EX03	`int Y::valeur_surprise() const noexcept {`
	`return -1 * valeur();`
	`}`		

Explications : ____

____

____

____

		Correct	Incorrect

EX04	`void Y::CalculerValeur(int v0, int v1) {`
	`SetValeur(v0 + 2 * v1);`
	`}`		

Explications : ____

____

____

____

EX05		Correct	Incorrect

	`void Y::CalculerValeur(int v0, int v1) {`
	`   SetValeurCru(v0 + 2 * v1);`
	`}`		

Explications : ______________________________

EX06		Correct	Incorrect

	`void Y::CalculerValeur(int v0, int v1) {`
	`   valeur_ = v0 + 2 * v1;`
	`}`		

Explications : ______________________________

EX07		Correct	Incorrect

	`int f0(const X &x) {`
	`   return x.valeur() + 10;`
	`}`		

Explications : ______________________________

EX08	Code	Correct	Incorrect

	`int f1(const X &x) {`
	`x.SetValeur(x.valeur () + 10);`
	`return x.valeur();`
	`}`		

Explications : ______________________________

____

____

____

EX09	Code	Correct	Incorrect

	`int f2(X &x) {`
	`x.SetValeur(x.valeur() + 10);`
	`return x.valeur();`
	`}`		

Explications : ______________________________

____

____

____

EX10	Code	Correct	Incorrect

	`int f3(X &x) {`
	`return x.valeur_surprise();`
	`}`		

Explications : ______________________________

____

____

____

		Correct	Incorrect
EX11	`int f4(const Y &y) {` `return y.valeur_surprise() * 5;` `}`		

Explications : ______

		Correct	Incorrect
EX12	`int f5(const Y &y) {` `return y.valeur() * 5;` `}`		

Explications : ______

		Correct	Incorrect
EX13	`void f6(Y &y) {` `y.SetValeur(5);` `}`		

Explications : ______

		Correct	Incorrect

EX14	`void f7(Y &y) {`
	`   y.CalculerValeur(2, y.valeur());`
	`}`		

Explications : ____________________

____

____

____

		Correct	Incorrect

EX15	`int f8(X &x) {`
	`   return 7 - f6(x);`
	`}`		

Explications : ____________________

____

____

____

		Correct	Incorrect

EX16	`int f9(Y &y) {`
	`   return 7 - f1(y);`
	`}`		

Explications : ____________________

____

____

____

Héritage privé

```
class Enfant
   : private Parent
{
   // ...
};
```

```
class Enfant : Parent {
   // ...
};
```

L'héritage privé repose sur l'emploi du mot clé `private` précédant le nom du parent lors de la déclaration de la classe enfant (voir à droite).

L'héritage privé est l'héritage par défaut de C++, même s'il s'agit d'une approche moins connue et moins comprise par la majorité que ne l'est l'héritage public. Ainsi, en l'absence de qualification dans une relation d'héritage dans lequel un enfant est une `class`[26], l'héritage sera privé.

On peut dire que l'héritage privé signifie qu'*`Enfant` dérive de `Parent`, et personne d'autre que lui ne le sait*. Le sens opérationnel de cette phrase est que seule la classe enfant peut tirer profit de l'information selon laquelle un `Enfant` est aussi un `Parent`.

Impact de cette affirmation : il est impossible pour un client de l'enfant de passer en paramètre un `Enfant` à tout sous-programme demandant qu'on lui passe un `Parent`.

La conversion d'enfant à parent, dans une situation d'héritage privé, existe mais est jugée *inaccessible*[27]. C'est d'ailleurs ce que le message d'erreur à la compilation vous indiquera si vous essayez de le faire.

> Il y a une exception: la conversion est valide si elle est faite par une méthode d'`Enfant`, puisque cette classe est *dans le secret des dieux*.
>
> On parle du descendant immédiat de `Parent` ici; les descendants indirects de `Parent` (au minimum, les enfants d'un enfant privé de `Parent`) n'auront pas ce privilège.

> Nous aborderons quelques applications concrètes de l'héritage privé et de l'héritage protégé dans la section ***Mieux exploiter le code existant***, plus bas.

Ceci signifie, par exemple, que si `Pikachu` hérite de manière privée de `Pichu`, alors un sous-programme prenant en paramètre une instance de `Pichu` ne pourra pas recevoir en paramètre une instance de `Pikachu` (car seul un `Pikachu` sait qu'il est aussi un `Pichu`). Il s'agit d'un secret bien gardé.

Illustration : si la classe `Pichu` existe, et si la classe `Pikachu` hérite de façon privée de `Pichu`, alors portez attention aux appels de fonction ci-dessous :

---

[26] Quand un enfant est un `struct`, l'héritage par défaut est public.

[27] … sauf si elle est réalisée dans une méthode de cette classe enfant. En effet, le code suivant est légal, mais contrevient en grande partie à la majorité des raisons pour lesquelles on pourrait avoir eu recours à l'héritage privé en premier lieu :

```
class Parent { /* ... */ };
class Enfant : private Parent { // le mot private est redondant ici
   // ...
public:
   // Conversions légales car réalisées dans une méthode de l'enfant.
   // Lui seul sait que la conversion est légale
   const Parent& GetParent() const noexcept { return *this; }
   Parent& GetParent() noexcept { return *this; }
};
```

```
// Fonction potentiel_electrique()
// Intrants: un Pichu
// Extrants: ---
// Rôle:     évalue le potentiel électrique de la créature en paramètre
double potentiel_electrique(const Pichu &);

// Fonction comparer_potentiels_electriques
// Intrants: un Pikachu et un Pichu
// Extrants: VRAI s'ils ont le même potentiel électrique, FAUX sinon
// Rôle:  tentera d'obtenir les potentiels électriques des deux créatures...
//        mais échouera (à la compilation!) lorsqu'elle essaiera de passer le
//        Pikachu (p0) en paramètre à la fonction potentiel_electrique(),
//        du fait que Pikachu est un héritier privé de Pichu
bool comparer_potentiels_electriques(const Pikachu &p0, const Pichu &p1) {
   const double PotentielPikachu  = potentiel_electrique(p0), // ILLÉGAL,
                PotentielPichu    = potentiel_electrique(p1); // OK;
   return PotentielPichu == PotentielPikachu; // == sur des double... pas gentil!
}
```

Héritage privé et membres du parent

L'utilité de l'héritage privé est de donner à la classe enfant ***et à elle seule, à l'interne***, accès aux membres publics et protégés de son parent.

Comme dans le cas d'héritage public, les membres privés du parent ne sont pas directement accessibles par l'enfant. Contrairement à l'héritage public, par contre, mis à part l'enfant lui-même, personne d'autre que l'enfant lui-même n'est autorisé à utiliser les membres publics ou protégés du parent à travers l'enfant.

Illustration : reprenons la classe `Pichu` et la classe `Pikachu`, et examinons la méthode `bzzz()` de la classe `Pichu` qui retourne une valeur sous la forme d'un nombre à virgule flottante. Un potentiel électrique, disons, pouvant par exemple être modifié par le niveau de fatigue de la petite bête.

Pichu.h	**Pikachu.h**
`class Pichu {`	
`// ...`	`class Pikachu : private Pichu { // redondant`
`double potentiel_;`	`// ...`
`// ...`	`public:`
`public:`	`double attaque_eclair() const;`
`double bzzz() const;`	`// ...`
`// ...`	`};`
`};`	

Supposons maintenant qu'on veuille implanter un sous-programme nommé `combat_eclair()` qui fasse combattre un `Pichu` et un `Pikachu`.

L'implantation proposée ci-dessous ne compilera pas : il est illégal, pour un sous-programme externe à la classe `Pikachu`, de se servir de la méthode `bzzz()` d'un `Pikachu` puisque le fait qu'un `Pikachu` est un `Pichu` n'est connu que des `Pikachu` eux-mêmes (même les `Pichu` n'en savent rien!).

```
void combat_eclair(const Pichu &p0, const Pikachu &p1) { // version 0
   const double AttaquePichu   = p0.bzzz(); // OK
   const double AttaquePikachu = p1.bzzz(); // ILLÉGAL (ne compile pas!)
   // ...
}
```

On le voit: *pris de l'extérieur*, à travers une instance de `Pichu`, on peut utiliser sans problème la méthode `bzzz()`, celle-ci étant publique. Par contre, pour une instance de `Pikachu`, héritant de manière privée de `Pichu`, on ne peut pas utiliser la méthode `bzzz()` et ce, bien que cette méthode existe.

Par contre, la classe Pikachu, à l'interne, sait très bien qu'elle hérite de la classe Pichu, et peut, elle, utiliser cette information à bon escient. Ainsi, elle peut (toujours à l'interne) avoir recours aux membres publics et protégés de Pichu, comme le démontre sa méthode attaque_eclair() ci-dessous :

```
double Pikachu::attaque_eclair() const {
   static const double MULTIPLICATEUR_PIKACHU = 10.0;
   return bzzz() * MULTIPLICATEUR_PIKACHU; // OK (utilisation interne)
}
```

On pourrait réécrire combat_eclair() pour utiliser la méthode attaque_eclair() d'un Pikachu et de la méthode bzzz() d'un Pichu. On obtiendrait à peu près ceci :

```
void combat_eclair(const Pichu &p0, const Pikachu &p1) { // version 1
   const double AttaquePichu   = p0.bzzz(); // OK
   const double AttaquePikachu = p1.attaque_eclair(); // OK
   // ...
}
```

Attention : l'idée ici n'est pas de dire qu'une forme d'héritage est meilleure que l'autre (ou l'inverse), mais bien de montrer l'impact de l'héritage privé sur l'accès aux membres d'une classe parent à partir d'une classe enfant.

Héritage privé et enrobage

On aurait pu faire fonctionner la version 0 de la fonction combat_eclair() en implantant une nouvelle méthode bzzz() à même la classe Pikachu. Cette méthode aurait, du point de vue d'un Pikachu, *caché* celle de son ancêtre Pichu.

Si cette méthode avait le même effet que la méthode attaque_eclair() ci-dessus, on aurait alors à peu près l'équivalent d'un enrobage comme on les voit par exemple parfois en procédant par composition.

Une difficulté technique s'impose par contre dans l'écriture, puisque si la méthode bzzz() d'un Pikachu veut utiliser la méthode bzzz() de son parent Pichu, *on ne peut plus l'appeler simplement* *bzzz()*[28].

[28] Il s'agit ici d'une très brève et très succincte introduction à la surcharge de méthodes (voir ***Héritage, surcharge de méthodes et espaces nommés***, plus loin).

La raison est que si la classe `Pikachu` a sa propre méthode `bzzz()`, lui faire appeler `bzzz()` sollicitera le `bzzz()` de `Pikachu`, qui s'appellerait alors lui-même, ce qui n'est pas ici l'effet désiré[29] :

```
double Pikachu::bzzz() const {
   static const double MULTIPLICATEUR_PIKACHU = 10.0;
   // VILAIN. bzzz() se rappelle elle-même... à l'infini!
   return bzzz() * MULTIPLICATEUR_PIKACHU;
}
```

Dans un tel cas, on veut spécifier explicitement que c'est du `bzzz()` d'un `Pichu` qu'on a besoin (le `Pichu` dont dérive le `Pikachu`, bien sûr).

Pour ce faire, on utilisera la syntaxe suivante :

```
double Pikachu::bzzz() const {
   static const double MULTIPLICATEUR_PIKACHU = 10.0;
   return Pichu::bzzz() * MULTIPLICATEUR_PIKACHU; // OK
}
```

Ceci a du sens parce qu'un `Pikachu` est un `Pichu`, et qu'il le sait!

Permettre à un enfant de référer explicitement à l'un de ses ancêtres nécessite, en C++, la même forme syntaxique que celle utilisée pour utiliser explicitement des membres de classe. Ceci n'entraîne pas d'ambiguïté, puisqu'il ne peut y avoir dans une même classe un membre d'instance et un membre de classe qui portent tous deux le même nom.

En Java, où les options d'héritage sont restreintes (héritage simple seulement une seule classe parent par classe enfant), on n'utilise pas le nom de la classe parent pour l'identifier, et on se limite plutôt à utiliser le mot clé `super` (p. ex. : `super.bzzz()`). Avec C++, la possibilité d'avoir plusieurs parents pour une même classe fait en sorte qu'un mot clé ne suffirait pas à déterminer le parent visé.

En C#, la situation est identique à celle rencontrée en Java, mais le mot clé à utiliser est `base` (p. ex. : `base.bzzz()`). En `VB.NET`, on utilisera `MyBase`.

[29] C'est parfois une bonne idée d'avoir un sous-programme qui s'appelle lui-même. On appelle cette manœuvre de la ***récursivité***. Peut-être êtes-vous déjà familière ou familier avec le sujet? Dans le cas de la méthode `bzzz()`, qui nous sert d'exemple ici, par contre, ça poserait de sérieux problèmes…

Notes philosophiques sur l'héritage privé

L'héritage privé, dans le concret, est souvent utilisé comme une alternative conceptuelle à la technique de composition.

L'héritage dénotant une relation sur le mode du verbe être (un `Is-A` *Relationship*), on fera appel à l'héritage privé dans une situation où l'enfant est vraiment un cas plus spécifique du parent, mais où on préfère quand même qu'il *apparaisse*, du point de vue du monde extérieur à l'objet lui-même, à la manière d'un composé contenant un parent.

Concevoir une classe par composition ou par héritage privé mène en général au même résultat du point de vue du code client, donc en s'en tenant strictement aux opérations possibles sur cette classe telle qu'elle est perçue par le monde extérieur. Sur le plan de la structure interne et de l'accès interne aux membres du parent, les deux approches ont leurs différences, visibles dans l'écriture des méthodes résultantes (l'enfant peut se traiter, à l'interne, comme un cas particulier de son parent).

Il est possible (et parfois souhaitable) de donner accès aux méthodes du parent à travers l'enfant en appliquant la technique de l'enrobage (le *Wrapping*, mentionné précédemment), mais les conséquences sont les mêmes que dans le cas de la composition.

En particulier, pour appliquer la technique de l'enrobage, il faut présumer connaître du parent *toutes* les méthodes qui seraient susceptibles d'apparaître comme utiles du point de vue d'un utilisateur de l'enfant; une connaissance exhaustive, donc, constituant une forme de bris discret d'encapsulation, dû au fait qu'un ajout au parent risque chaque fois de demander une modification de l'enfant. Cette technique tend donc à mieux fonctionner lorsque les gens œuvrant sur la classe enfant ont un certain contrôle sur l'évolution de la classe parent (ce qui est vrai, en général, pour l'ensemble des techniques d'enrobage et de délégation).

Certains iront jusqu'à prétendre que, à toutes fins pratiques, l'héritage privé n'est pas tant de l'héritage qu'une composition déguisée, et que seul l'héritage public est un véritable héritage. C'est d'ailleurs sûrement vrai en Java et dans les langages `.NET` où les concepteurs ont choisi de n'offrir aucune forme alternative à l'héritage public.

En fait, la réalité est un peu plus complexe que cela : par héritage privé, on obtient une forme particulière d'***héritage d'implémentation***, un cousin de la composition et de l'***héritage d'interface***, mais avec un subtil glissement sémantique. Nous reviendrons plus loin sur ces deux idées, qui sont des visions distinctes et des applications différentes du concept d'héritage, et dont l'emploi est du ressort direct des décideurs d'une équipe de développement.

> Philosophiquement, un objectif de saine programmation `OO` est de réduire le couplage entre les entités des programmes, en particulier entre les classes. Une manière simple de réduire le couplage est de réduire la surface publique des objets, en particulier de garder au minimum le nombre de membres publics d'un objet.
>
> L'héritage privé s'inscrit dans cette veine : une relation entre deux objets qui n'est connue que de l'enfant est une relation de plus faible couplage que ne le serait une relation connue de tous les objets comme l'est l'héritage public.
>
> Si les concepteurs d'un système désirent modifier une relation construite sur l'héritage privé, l'impact de ce changement de design sera localisé dans la classe enfant. Au contraire, une relation d'héritage public est connue de tous et peut être exploitée par tous (tous peuvent considérer l'enfant comme un cas particulier de son parent), et modifier une telle relation n'est presque pas possible, le couplage étant trop fort et les conséquences sur le code client étant trop grandes.

Exercices – Série 01.1

Présumant les classes suivantes :

```
class X {
   int valeur_;
public:
   X();
   ~X();
   int valeur() const noexcept;
   void SetValeur(int);
protected:
   void SetValeurCru(int);
};
```

```
#include "X.h"
class Y : private X {
public:
   Y();
   ~Y();
   int valeur_surprise() const noexcept;
   void CalculerValeur(int, int);
};
```

Indiquez, pour chaque sous-programme ci-dessous, s'il est correctement écrit ou non (on entendra ici *correct* au sens de *n'enfreignant pas les règles du langage*), et expliquez pourquoi.

EX00

	Correct	Incorrect

`int X::valeur() const noexcept {`
`   return valeur_;`
`}`		

Explications : ______________________________

____

____

____

EX01

	Correct	Incorrect

`void X::SetValeur(int val) {`
`   if (val > 0) {`
`      SetValeurCru(val);`
`   }`
`}`		

Explications : ______________________________

____

____

____

EX02

void X::SetValeurCru(int val) {
   valeur_ = val;
}
	Correct	Incorrect

Explications : ____

____

____

____

EX03

int Y::valeur_surprise() const noexcept {
   return -1 * valeur();
}
	Correct	Incorrect

Explications : ____

____

____

____

EX04

void Y::CalculerValeur(int v0, int v1) {
   SetValeur(v0 + 2 * v1);
}
	Correct	Incorrect

Explications : ____

____

____

____

		Correct	Incorrect

EX05	`void Y::CalculerValeur(int v0, int v1) {`
	`   SetValeurCru(v0 + 2 * v1);`
	`}`		

Explications : ______________________________

____

____

____

		Correct	Incorrect

EX06	`void Y::CalculerValeur(int v0, int v1) {`
	`   valeur_ = v0 + 2 * v1;`
	`}`		

Explications : ______________________________

____

____

____

		Correct	Incorrect

EX07	`int f0(const X &x) {`
	`   return x.valeur() + 10;`
	`}`		

Explications : ______________________________

____

____

____

		Correct	Incorrect

EX08	`int f1(const X &x) {`
	`   x.SetValeur(x.valeur() + 10);`
	`   return x.valeur();`
	`}`		

Explications : ______________________________

____

____

____

		Correct	Incorrect

EX09	`int f2(X &x) {`
	`   x.SetValeur(x.valeur() + 10);`
	`   return x.valeur();`
	`}`		

Explications : ______________________________

____

____

____

		Correct	Incorrect

EX10	`int f3(X &x) {`
	`   return x.valeur_surprise();`
	`}`		

Explications : ______________________________

____

____

____

		Correct	Incorrect

EX11	`int f4(const Y &y) {`
	`   return y.valeur_surprise() * 5;`
	`}`		

Explications : ____________________

____

____

____

		Correct	Incorrect

EX12	`int f5(const Y &y) {`
	`   return y.valeur() * 5;`
	`}`		

Explications : ____________________

____

____

____

		Correct	Incorrect

EX13	`void f6(Y &y) {`
	`   y.SetValeur(5);`
	`}`		

Explications : ____________________

____

____

____

		Correct	Incorrect

EX14	`void f7(Y &y) {`
	`   y.CalculerValeur(2, y.valeur());`
	`}`		

Explications : ____________________

____

____

____

		Correct	Incorrect

EX15	`int f8(X &x) {`
	`   return 7 - f6(x);`
	`}`		

Explications : ____________________

____

____

____

		Correct	Incorrect

EX16	`int f9(Y &y) {`
	`   return 7 - f1(y);`
	`}`		

Explications : ____________________

____

____

____

Héritage protégé

Vu de l'extérieur, l'héritage protégé a le même impact que l'héritage privé, au sens où il donne à l'enfant un accès à la fois aux membres protégés et publics du parent, sans toutefois donner cet accès au monde extérieur.

Cependant, contrairement à l'héritage privé, *l'héritage protégé donne aussi accès à cette information aux descendants de l'enfant*.

L'héritage protégé est l'équivalent OO d'un secret de famille.

L'héritage protégé repose sur l'emploi du mot clé `protected` précédant le nom du parent lors de la déclaration de la classe enfant (voir l'exemple à droite).

```
class Enfant
   : protected Parent
{
   // ...
};
```

On peut dire que l'héritage protégé signifie qu'*`Enfant` dérive de `Parent`, et seuls lui et ses propres descendants le savent*. Le sens opérationnel de cette phrase est que la classe enfant et ses propres descendants peuvent tirer profit de l'information selon laquelle un `Enfant` est aussi un `Parent`, alors que le reste du code client ne le peut pas.

Les règles relatives à l'accès aux membres d'un parent par un enfant provenant d'une relation par héritage protégé sont les mêmes que celles applicables dans le cas d'une relation d'héritage privé, à la différence près que la connaissance du lien entre ce parent et cet enfant se perpétue chez les descendants[30] de l'enfant.

Nous aborderons quelques applications concrètes de l'héritage privé et de l'héritage protégé dans la section ***Mieux exploiter le code existant***, plus bas.

[30] Descendants publics ou protégés, bien sûr. L'introduction d'une catégorie de spécification d'accès plus contraignante (privé étant plus contraignant de protégé, et protégé étant plus contraignant que public) restreint *de facto* les droits pour la descendance subséquente en entier.

Résumé des spécifications de sécurité et d'héritage

Il est de rigueur de faire un résumé tabulaire des informations contenues dans les pages qui ont précédé, car il y en a beaucoup! Voici, donc, un bref de ce que nous venons de couvrir quant aux spécifications de protection et de sécurité appliquées aux membres et à l'héritage.

Membre

Privé	Protégé	Public
Un membre **privé** d'un parent n'est connu que de lui, et est réservé pour usage interne.	Un membre **protégé** d'un parent n'est connu que de lui, de ses enfants directs et (peut-être) de ses descendants, mais pas d'autres sous-programmes ou d'autres objets.	Un membre **public** d'un parent est connu de lui, de ses enfants et (peut-être) de ses descendants, et est dévoilé au monde extérieur (autres sous-programmes, autres objets), servant ainsi d'interface pour l'objet qui en est propriétaire.

Héritage

Privé	Protégé	Public
L'enfant est le seul à savoir qu'il est un enfant de son parent. Ni le monde extérieur (autres objets, autres sous-programmes), ni ses propres enfants ne le savent.	L'enfant et ses propres descendants sont les seuls à savoir qu'il est un enfant de son parent. Le monde extérieur (autres objets, autres sous-programmes) n'a pas accès à cette information.	L'enfant sait, comme tout le monde, qu'il est un enfant de son parent.
L'enfant pourra accéder directement aux membres publics et protégés de son parent, mais ses propres enfants ne le pourront pas.	L'enfant pourra accéder directement aux membres publics et protégés de son parent, ses propres enfants pourront en faire autant, mais le monde extérieur ne le pourra pas.	Cet enfant pourra accéder directement aux membres publics et protégés de son parent, ses propres enfants pourront en faire de même, et le monde extérieur pourra utiliser les membres publics du parent à travers l'enfant.
Seul l'enfant lui-même pourra se traiter comme un cas particulier de son parent.	Par ses propres méthodes, l'enfant pourra se traiter comme un de ses parents, et ses propres enfants pourront en faire autant, mais – encore une fois – le monde extérieur ne le pourra pas.	Tous pourront traiter l'enfant comme un cas particulier de son parent.

Dans d'autres langages

Les stratégies d'héritage privé et protégé n'existent pas dans les langages Java, C# et VB.NET. Dans ces trois langages, *hériter* signifie *hériter publiquement* et l'immense majorité des programmeuses et des programmeurs ne sont pas même au courant que ce ne soit là qu'une option parmi plusieurs.

Des nuances quant aux qualifications d'encapsulation applicables à l'héritage intéressent un nombre restreint d'individus, la plupart étant impliquées dans le développement de bibliothèques ou d'infrastructures OO sophistiquées. Cela dit, pour ces personnes, la possibilité de manipuler finement l'état ontologique d'un objet et des relations entre classes est fort appréciable.

En général, les gens présentent l'héritage protégé (et, dans une moindre mesure, l'héritage privé) comme un élément de fondation. Si l'héritage public est une relation du type Est un(e) (*Is-A*) et si la composition et l'agrégation sont des relations du type Possède un(e) (*Has-A*), il se trouve que les relations d'héritage privé et protégé sont des relations du type Est implémenté(e) selon (en anglais : *Is-implemented-in-terms-of*).

Plusieurs exemples typiques d'héritage privé et protégé dans la littérature mettent en relief des conteneurs qui ne sont pas destinés à un usage polymorphique. L'exemple classique propose de représenter à l'interne un ensemble comme une sorte de liste et un tableau associatif comme une sorte d'ensemble sans nécessairement vouloir traiter de manière polymorphique les ensembles et les tableaux associatifs comme des listes, donc faire en sorte que ce choix d'implémentation soit *véritablement* un choix d'implémentation, fait à l'insu du code utilisateur.

Gardons en tête ici que toute information publique sur un objet est susceptible d'être utilisée par un éventuel client, et doit conséquemment être tenue à jour.

S'il s'avère qu'un ensemble est construit comme un cas particulier de liste *mais que ce choix est susceptible de changer avec le passage du temps*, alors il est préférable de ne pas construire de code client reposant sur ce savoir et, conclusion logique, de cacher ce fait au client. En général, mieux vaut ne révéler aux clients que ce qui risque de survivre au passage du temps.

Qu'un ensemble ait été pensé comme un cas particulier de liste est ici un point de gestion interne, et l'héritage privé est tout désigné.

> En Java, C# et VB.NET, l'héritage privé ou protégé se simule, au besoin, par composition ou agrégation et en enrobant les méthodes de l'objet englobé par des méthodes dans l'objet englobant. Quand un concept n'est pas intégré à un langage, cela dit, les gens n'ont pas souvent le réflexe d'y avoir recours, ce qui explique que les programmeuses et les programmeurs utilisant ces langages tendent à chercher d'autres approches pour solutionner des problèmes auxquels l'héritage privé ou protégé pourrait s'appliquer.
>
> Notons que, si l'objet englobé dans une relation d'enrobage et de délégation possède des membres protégés, un dérivé protégé ou privé y aurait accès alors qu'un objet englobant au sens de la composition ou de l'agrégation ne pourra pas y accéder. Dans une telle situation, avec Java, C# ou VB.NET, le seul recours est d'ouvrir la barrière d'encapsulation plus que désiré et révéler ce membre comme public.
>
> La solution sans héritage privé ou protégé réduit alors la qualité du design OO.

Mieux exploiter le code existant

Une situation que les programmeurs C++ rencontrent plus souvent que les programmeurs Java, C# ou VB.NET est celle où du code existant, souvent ancien et développé à l'aide de stratégies qui ne sont pas OO le moins du monde, doit être intégré dans un système qui, lui, se veut OO. Les programmeurs .NET et Java, dans la majorité des cas, utiliseront une bibliothèque de classes volumineuse qui enrobe des fonctionnalités plus primitives et les intègre au modèle privilégié par leur plateforme (la JVM ou le CLR) et, quand leurs besoins ne sont pas couverts par les classes déjà disponibles, en viendront souvent à interfacer avec du code C ou C++ (ce qui se fait, mais constitue une opération délicate).

Le cas type d'une situation d'intégration de code procédural dans un programme OO est celui de l'intégration entre C et C++. Les struct du langage C sont des enregistrements au sens classique de la programmation structurée, soit des agrégats de données pouvant être de types différents, sans la moindre méthode. Ces types portent un surnom en C++ : on dit d'eux que ce sont des types POD, pour *Plain Old Datatypes*.

Les bibliothèques standards du langage C regorgent de POD très utiles, autour desquels s'articulent tout un tas de fonctions puissantes, raffinées et ayant survécu au passage du temps. Vouloir les exploiter dans un contexte OO est tout à fait raisonnable et défendable.

Une telle situation, remarquée par un dénommé ***David R. Tribble***[31], pourrait survenir dans un cas où on reconnaîtrait l'intérêt de construire une classe de représentation du temps, avec services en prime, autour du type enregistrement struct tm du langage C.

Le code ressemblerait à celui proposé à droite.

```
#include <ctime>      // std::tm, un struct
class Temps : protected std::tm {
public:
   // méthodes encapsulant l'accès à std::tm
};
```

Remarquez, dans ce cas précis, le choix d'un mode d'héritage protégé. Ceci signifie que le savoir qu'une instance de Temps est aussi un std::tm sera propagé aux instances des classes dérivées de Temps, ce qui est un choix de design parmi d'autres.

Choisir l'héritage public dans ce cas-ci aurait permis à tout code utilisateur de prendre une référence sur une instance de Temps et d'en manipuler les attributs directement, ce qui constituerait un bris d'encapsulation direct. Dans ce cas, clairement, l'héritage public ne serait pas un choix recommandable.

Choisir l'héritage privé aurait empêché les descendants de la classe Temps de profiter du savoir selon lequel une instance de Temps est aussi un std::tm et aurait donc empêché les instances d'une classe dérivée de la classe Temps d'utiliser lui-même des fonctions C sur ses propres attributs std::tm. Ce choix peut se défendre; c'est une question d'optique.

[31] Voir http://cpptips.hyperformix.com/cpptips/prot_inher2

Un cas (simplifié) où l'héritage privé pourrait être indiqué serait celui où il existe une classe commerciale qui fait tout ce qu'on souhaite… et même un peu trop. La classe en question expose peut-être un attribut public, brisant l'encapsulation, ou révèle peut-être une méthode dans son interface publique qui, selon notre point de vue, aurait dû demeurer privée.

Par exemple, imaginons la classe `Compte` dont une ébauche est proposée à droite, et présumons que cette classe fasse tout ce qu'on souhaite d'un bon compte bancaire, incluant offrir une gamme de services protégés qui sont utiles au développement de catégories plus spécialisées de comptes bancaires, mais qu'elle expose aussi une méthode publique `SetSolde()` qui, à notre avis, ne devrait pas faire partie du visage public d'un compte bancaire.

```
class Compte {
   // ... trucs privés et protégés utiles
public:
   void SetSolde(float);
   float solde() const noexcept;
   // ...
};
```

Si nous souhaitions utiliser `Compte` (pour éviter de réécrire la logique toute entière qui nous est tombée du ciel avec cette classe) pour définir nos propres comptes bancaires, nous pouvons procéder par enrobage et délégation… sauf si nous comptons aussi tirer profit des outils protégés de `Compte`, car seule sa descendance a accès à ses membres protégés.

Pour faire disparaître `SetSolde()` du portrait, on devra alors créer un dérivé privé de `Compte` et déléguera les méthodes à révéler au monde vers les méthodes de son parent (en omettant volontairement d'exposer `SetSolde()`, dont nous ne voulons pas). `Compte` sera « enfoui » dans notre classe et elle seule saura qu'elle dérive de `Compte`. De cette manière, notre classe exposera à peu de frais la gamme de services de `Compte` qu'elle juge pertinente, et pourra malgré tout utiliser les services protégés de `Compte` auxquels elle tenait.

Sans héritage privé, nous devrions :

- soit utiliser l'héritage public et exposer `SetSolde()` au code client, un désastre;
- soit utiliser l'enrobage et la délégation, donc nous priver des membres protégés de `Compte`;
- soit travailler un peu plus fort et définir dans notre classe une classe interne et privée, dérivant publiquement de `Compte` mais inconnue du code client de notre propre classe, puis faire en sorte que les services de notre classe délèguent aux services de la classe interne. Beaucoup de travail et une double délégation pour chaque invocation de service.

Pouvoir dériver de manière protégée ou privée permet d'intégrer de manière simple et sécurisée des entités précédant le modèle `OO` dans un monde `OO`. Par son choix de mode d'héritage, la classe `Temps` nous dit *Je suis un* `std::tm`*, soit*[32]*, mais cela ne vous concerne pas*. Le dérivé protégé ou privé construit une façade sécurisée autour d'une représentation primaire et permet d'ajouter des services plus sophistiqués autour de code existant, tout en maintenant l'identité propre de l'objet en cours de définition.

---

[32] Remarquez ici que le *Je suis un* `std::tm` est important puisque cela justifie l'emploi d'héritage. Dans le cas où l'idée à exprimer n'est pas basée sur le verbe être, la composition et l'agrégation sont habituellement de bien meilleures options.

Héritage, surcharge de méthodes et espaces nommés

La tradition C++ a été bousculée, à certains égards, avec l'arrivée de la norme ISO du langage en 1998. L'un des lieux où cette bousculade a apporté les changements les plus subtils est dans l'introduction des espaces nommés[33] (**namespace**), et dans la relation entre ce nouveau concept et les classes elles-mêmes.

La norme ISO de C++ a presque complètement éliminé les déclarations vraiment globales, en introduisant le concept d'espace nommé, regroupant logiquement des types, classes, fonctions, variables, *etc.* selon la vocation de chaque espace.

Les éléments jugés standard par la norme ISO du langage sont déposés dans l'espace nommé std, ce qui nous demande de préfixer le nom de ces éléments par **std::**. On ne peut pas ajouter des éléments à cet espace nommé (il appartient aux gens qui gèrent le standard du langage).

Des éléments d'un hypothétique espace nommé POO devraient, conséquemment, être préfixés par POO::.

On crée un espace nommé comme on crée une classe ou un enregistrement (struct), comme le montre l'exemple à droite.

Un espace nommé (outre std) est un regroupement ouvert. On peut y ajouter des éléments provenant de plusieurs fichiers, comme bon nous semble.

```
// ici, notre namespace n'existe pas
namespace POO {
   // maintenant, il existe!
   int f(int);
   class X {
      int val_;
   public:
      X();
      int valeur() const noexcept;
   };
};
//
// Les définitions vont habituellement
// dans un fichier source
//
int POO::f(int x) {
   return x + 123;
}
// constructeur par défaut de POO::X
POO::X::X() : val_{} {
}
int POO::X::valeur() const noexcept {
   return val_;
}
```

Lorsqu'un programme ne compte utiliser qu'un seul espace nommé, il peut appliquer la commande using namespace suivi du nom de l'espace nommé désiré. On le fait souvent avec l'espace nommé std pour faciliter le bon fonctionnement du code pré-ISO, et pour alléger la syntaxe lors d'une première approche au langage. Évitez toutefois d'agir ainsi dans un .h, puisque vous affecterez alors tous les fichiers qui l'incluront.

Dans la mesure du possible, réduisez les alias obtenus par using au minimum pour éviter de polluer vos programmes. Si vous n'utilisez de l'espace std que std::cout et std::endl, alors préférez

```
using std::cout;
using std::endl;
```

…au (bien trop large, mais peut être acceptable pour vos propres .cpp)

```
using namespace std;
```

[33] Certains utiliseront le vocable ***espace de noms***, qui est aussi raisonnable.

Lorsqu'on ne veut que certains éléments d'un espace nommé, on peut les énumérer un à un, comme dans l'exemple à droite.

```
#include <iostream>
int main() {
   using std::cout;
   using std::endl;
   cout << "Allo!" << endl;
}
```

Utiliser `using namespace XYZ` fait en sorte d'utiliser, par défaut, les éléments de l'espace nommé `XYZ`, *quels qu'ils soient*. Dans l'usage courant, on privilégiera les applications plus sélectives de `using`.

Les espaces nommés aident à organiser de manière plus rigoureuse les classes, sous-programmes et autres éléments de programmation. L'habitude que nous avons de négliger la mention d'espaces nommés dans notre code devrait, avec le temps, disparaître.

Dans un programme C++, les entités explicitement déclarées dans un espace nommé anonyme n'apparaissent pas à l'édition des liens. Ceci permet de remplacer le mot clé `static` du langage C lorsque celui-ci est appliqué à une entité globale par l'insertion de cette entité dans un espace nommé anonyme.

À bien des égards, on pourrait remplacer de la gestion de versions et de produits de firmes concurrentes (vérifiées par des humains) par de la gestion d'espace nommé qui, elle, est vérifiable (et vérifiée) par un compilateur.

Ajouter à un espace nommé

Les espaces nommés sont ***ouverts pour ajouts***. Ceci inclut l'espace nommé `std`, bien qu'il ne soit pas ouvert aux pauvres mortels que nous sommes. On peut ajouter à notre guise des éléments à un espace nommé existant.

```
// présumant POO déjà déclaré
namespace POO { // on y ajoute g()
   float g(); // déclaration
};
// ...
float POO::g() { // définition
   return 1.5f;
}
```

Ajouter un élément à un espace nommé se fait de la même manière que lorsqu'on crée un tel espace (voir à droite). Ainsi, pas besoin de créer explicitement un tel espace; il sera créé dès qu'on lui ajoutera un premier élément.

Cette ouverture pour ajouts est très utile en pratique. Par exemple, `<iostream>` définit les classes d'entrées/ sorties standards et les place dans `std`. De son côté, `<string>` définit les chaînes de caractères standards et les place aussi dans `std`.

Un fichier source peut inclure l'un, l'autre ou les deux fichiers d'en-tête à sa guise, introduisant dans `std` un nombre de symboles qui variera selon ses besoins, et le compilateur n'aura qu'à se préoccuper que ce dont le programme se sert vraiment.

```
int f(int); // global
#include <iostream>
int main() {
   using std::cout;
   using std::endl;
   cout << f(4)   // implicite
        << endl
        << ::f(4) // explicite
        << endl;
}
```

L'espace nommé anonyme

Lorsqu'un élément de langage est déclaré *globalement,* comme le sont les classes et les sous-programmes de ce cours jusqu'ici, cet élément est en fait insérés dans un espace nommé anonyme, et peut être préfixé de `::` tout simplement.

Question : quel est le nom complet de `main()`?

```
// Ces fonctions C++ sont
// toutes différentes
int f(int);
int f();
int f(float);
class X {
public:
   // toutes ces méthodes
   // sont différentes
   X();
   X(int);
   int f(int);
   int f(int) const;
   int f(double);
};
```

Surcharge et dissimulation de noms

Là où les espaces nommés interfèrent dans notre réflexion est dans la gestion, par le compilateur, de la ***surcharge des méthodes***.

La **surcharge de sous-programmes** est un concept près des concepts `OO`. L'idée est de permettre plusieurs sous-programmes portant le même nom, dans la mesure où ils diffèrent autrement l'un de l'autre[34].

La **surcharge de méthodes à l'intérieur d'une même classe** est naturelle pour qui a compris le mécanisme de la construction.

Pour une même classe `X`, on peut très bien avoir plusieurs constructeurs, donc plusieurs méthodes `X::X()`, dans la mesure où ils ont chacune une liste de paramètres différente.

```
class A {
public:
   void f();
   void f(int);
};
class B : public A {
};
int main() {
   B b;
   b.f(); // légal
}
```

Lorsque l'héritage entre en jeu, la mécanique se complique. Par exemple, dans le cas visible ici, l'instance `b` déclarée dans `main()` a accès aux méthodes `f()` et `f(int)` de `B`, qui les hérite publiquement de `A`.

Ceci est naturel, et découle des règles usuelles de l'héritage public. En effet, si `B` dérive publiquement de `A`, alors il devrait offrir publiquement toutes les méthodes publiques de `A` à ses utilisateurs externes.

---

[34] **Rappel** : en C++, cette différenciation se fait sur la base des noms des sous-programmes, puis par le nombre de paramètres des sous-programmes, puis par le type d'au moins un paramètre des sous-programmes, et enfin par la présence ou l'absence par des qualifications `const` et `volatile` (dans le cas des méthodes d'instance).

Dans l'extrait à droite, par contre, l'appel à `b.f()` est ***illégal***, ce qui va, pour bien des gens, un peu à l'encontre de l'intuition. Pour comprendre la raison de cette illégalité, il faut comprendre la relation entre classe et espace nommé.

```
class A {
public:
   void f();
   void f(int);
};
class B : public A {
public:
   // surcharge de A::f(int)
   void f(int);
};
int main() {
   B b;
   b.f(); // illégal!?!?!
}
```

Il se trouve que, dans un souci d'homogénéité, les concepteurs de C++ ont cherché à fondre structurellement les concepts de classe et d'espace nommé en une seule et même entité conceptuelle[35]. Faire des espaces nommés un cas particulier de classe aurait posé problème, du fait que les classes on par défaut des membres privés et que cela aurait signifié relâcher, pour un cas particulier de classe, les règles de sécurité[36].

Alors ils ont fait le contraire, et fait de la classe une catégorie spéciale (et *instanciable*) d'espace nommé. Cela explique d'ailleurs en partie les ressemblances syntaxiques et lexicales entre ces deux entités.

Pourquoi, maintenant, le code ci-dessus pose-t-il problème? Il se trouve que, en C++, lorsque deux déclarations de même nom existent ***à un même niveau de portée***[37], le compilateur peut les distinguer l'une de l'autre à l'aide des règles de surcharge énoncées plus haut.

Par contre, lorsque deux sous-programmes de même nom existent ***à deux niveaux de portée différents***, comme dans le cas ci-dessus, alors *le niveau de portée le plus local domine toujours, et cache en totalité le niveau de portée le moins local dès qu'un conflit de nom se produit*. Cette dissimulation se fait non pas sur la base de la signature mais bien sur la base du nom.

Dans notre cas, le conflit se situe entre `B::f(int)` et `A::f(int)`, et fait en sorte que, d'une instance de `B`, l'existence même de méthodes nommés `f()` de `A` sont cachées, *peu importe leurs paramètres*. L'intuition aurait suggéré que seule `A::f(int)` aurait été caché ici, or ce n'est pas le cas du tout.

[35] L'espace nommé au sens strict (`namespace`) est en fait très près des classes, à ceci près qu'une classe est *fermée pour ajout* (on ne peut y ajouter des membres une fois ses déclaration complétée) et instanciable, alors qu'un espace nommé est *ouvert pour ajout* (on peut lui ajouter des membres à loisir) et non instanciable.

[36] Les `struct` font exception à cette démarche, entre autres pour des raisons historiques liées à la compatibilité avec le langage C et les bibliothèques existantes.

[37] Deux méthodes d'une même classe, par exemple, ou encore deux sous-programmes *globaux*, donc dans l'espace nommé anonyme.

Solutionner ce problème implique de reconnaître, à même B, qu'on peut accepter d'appeler à travers lui les méthodes de A malgré la présence d'un conflit de nom, donc exposer à travers B certains noms de A qui, autrement, demeureraient cachés.

L'ajout à faire dans la classe enfant (ici, la classe B) pour y arriver est :

```
using A::f;
```

c'est-à-dire (par défaut) exposer, à travers toute instance de B, les méthodes d'instance du parent A, en privilégiant bien sûr les noms les plus locaux. À travers un B, par exemple, on obtiendra la méthode `B::f(int)` mais on obtiendra `A::f()`.

Si aucun conflit de nom ne survenait (par exemple, si les méthodes de B s'appelaient `g()` plutôt que `f()`), alors on n'aurait pas besoin de la mention `using` du tout. Cette mention sert à ouvrir l'accès (normalement clos) à un niveau de portée autrement bloqué; d'aiguiller les appels vers un niveau hiérarchique inhabituel.

```
class A {
public:
   void f();
   void f(int);
};
class B : public A {
public:
// autoriser de chercher les méthodes
// nommées f dans le parent A (à part
// la méthode f(int), bien sûr, qui doit
// être prise dans B
   using A::f;
   void f(int); // surcharge de A::f(int)
};
int main() {
   A a;
   B b;
   a.f();  // appelle le f du A dans a
   a.f(3); // appelle le f du A dans a
   b.f();  // appelle le f du A dans b
   b.f(3); // appelle le f du B dans b
}
```

Dans d'autres langages

Les espaces nommés sont véritablement arrivés dans le monde C++ sur le tard, soit près de quinze ans suivant le début de l'utilisation réelle et (éventuellement) massive de ce langage. Leur analogue le plus direct en Java est le paquetage (*package*), alors que dans les langages `.NET` on a les regroupements logiques par espaces nommés et par assemblages (*assembly*).

Un **assemblage** `.NET` est une unité compilée en un format intermédiaire (le format `CIL`) du modèle `.NET`, typiquement une `DLL` ou un `EXE` destinés à être pris en charge par le `CLR`.

Notez que Java et C# n'admettent pas de fonctions globales et n'ont pas le concept d'espace nommé anonyme dont C++, pour des raisons de tradition, doit s'accommoder.

Divers langages, comme Pascal ou Modula, utilisent depuis longtemps des regroupements logiques par modules, mais l'héritage C de C++ a fait en sorte que l'inclusion lexicale des fichiers d'en-tête (directives `#include`) et la compilation séparée des unités de traduction est longtemps demeurée (outre les classes elles-mêmes) le seul réel regroupement de code aux yeux du compilateur[38]. L'injection somme toute tardive d'espaces nommés dans C++ explique à la fois la souplesse du compilateur envers leur application et leur caractère optionnel.

Les langages plus récents (et qui ne traînent pas avec eux les traditions du langage C) comme Java, C# et `VB.NET` ont intégré le découpage logique du code en modules dès le début, chacun à sa manière. Ceci explique à la fois le caractère plus strict avec lequel ces langages traitent les regroupements logiques et le rôle que ceux-ci jouent dans les programmes, du point de vue ses clauses de sécurité et de l'encapsulation.

Pour tracer un comparatif technologique et philosophique, rappelons qu'**en C++**, un espace nommé est une unité de regroupement logique, ouverte pour ajout, mais pas une unité de sécurité.

Le caractère ouvert pour ajout d'un espace nommé implique qu'il soit possible d'y ajouter des éléments au besoin; le fait que l'espace nommé n'y soit pas une unité de sécurité implique que toute entité logée dans un espace nommé est accessible à tout client ayant accès à l'espace nommé selon les mêmes règles que celles applicables normalement aux entités globales. En retour, une classe est fermée pour ajout : une fois que le `;` fermant sa déclaration a été atteint par le compilateur, il est impossible d'ajouter des membres à une classe.

On ne peut pas, en C++, être un membre privé ou protégé d'un espace nommé. Tout ce qui est déclaré dans un espace nommé y est public, accessible à tout code client l'ayant inclus. Toutes les classes sont des idées pures (elles sont toutes statiques).

[38] Notez bien que le découpage plus *physique* des modules objets, par exemple en bibliothèques à liens statiques ou dynamiques, fait partie du portrait depuis longtemps mais n'est pas en soi une particularité d'un langage ou de l'autre et a plus à voir avec l'éditeur de liens qu'avec le compilateur.

En Java, l'équivalent d'un espace nommé est un paquetage. Un paquetage regroupe des classes et des interfaces[39] ayant un lien entre elles en un ensemble, un peu comme on le fait avec des espaces nommés, des bibliothèques et des modules dans d'autres langages. Une classe peut y être statique ou non.

En Java, les classes font partie de paquetages et peuvent se voir apposer des qualifications de sécurité. Ces qualifications ont un impact sur l'identité et la provenance des clients potentiels d'une classe donnée. Ces qualifications sont au nombre de quatre : privé, privé au paquetage, protégé et public. Le niveau de protection le plus strict pour une classe est `private`. Apposer la qualification `private` sur une classe empêche même les classes du même paquetage (mais dans un autre fichier) d'y avoir accès. La classe devient un détail d'implémentation, dont les instances sont vouées à être utilisées de manière indirecte[40].

Si on ne lui appose pas de qualification explicite, une classe est dite ***privée au paquetage*** (*Package Private*)[41], ce qui signifie qu'elle n'est pas accessible hors de son paquetage. Il n'est pas possible de qualifier explicitement, par programmation, une classe comme étant privée au paquetage. Cette qualification de sécurité pour l'accès à une classe ne peut être indiquée que de manière implicite, par absence de toute autre qualification.

Une classe qualifiée `protected` est connue d'elle-même, *des autres classes de son paquetage* et de sa descendance. Visiblement, la qualification `protected` révèle plus (et encapsule moins rigoureusement) en Java qu'en C++.

Le fait que `protected` donne en Java un privilège d'accès aux autres entités du paquetage et que l'absence de qualification corresponde à une qualification implicite *privé au paquetage* met en relief l'importance d'y privilégier l'emploi explicite de la qualification `private` : l'ouverture implicite de la barrière de protection d'un objet aux entités partageant son paquetage complique, à moyen terme, l'entretien du code Java.

Enfin, une classe qualifiée `public` est connue de tous, mais une seule classe qualifiée `public` est permise par unité de traduction (par fichier `.java`), et le nom du fichier dans lequel apparaîtra la classe publique `X` doit absolument être `X.java`.

Un paquetage Java peut être importé en bloc ou en partie par du code utilisateur. L'exemple à droite importe la classe `IOException` du paquetage `java.io`, mais aurait pu importer toutes les classes de ce paquetage en important `java.io.*`, ou ne rien importer en explicitant, dans l'instruction `catch`, que l'exception attrapée est une `java.io.IOException`.

```
import java.io.IOException;
public class Test {
   public static void main(String [] args) {
      try {
         int val = System.in.read();
      } catch (IOException ex) {
         // gestion du problème
      }
   }}
```

[39] Voir la section ***Héritage d'interfaces***, plus loin.

[40] Nous reviendrons sur ces techniques à partir des sections ***Polymorphisme*** et ***Méthodes abstraites*** et ***Classes abstraites***, plus loin.

[41] Voir [JavaAcCtl] pour plus de détails. Vous remarquerez probablement que le code en exemple pour présenter le lien entre une classe dérivée et le mot clé `protected` est accompagné de commentaires incorrects.

L'importation de paquetages Java est en fait une importation de noms, semblable aux directives `using` de C++. Les noms peuvent être utilisés sans importation par un programme dans la mesure où ils sont spécifiés en entier; importer un nom permet d'en abréger l'écriture.

Ce qui permet à un compilateur Java de retrouver un nom par sa qualification complète est que les paquetages Java correspondent à une structure de répertoire, dont la racine logique est déterminée par une variable d'environnement nommée `CLASSPATH`. Ainsi :

- toutes les classes d'un même répertoire seront habituellement dans le même paquetage; et
- un répertoire contenant un paquetage peut avoir des sous-répertoires qui, sur le plan logique, sont des « sous-paquetages »;
- importer la totalité d'un paquetage (avec le suffixe `.*`) n'inclut toutefois pas récursivement les sous-paquetages. Ainsi, `import java.awt.*;` n'entraîne pas l'inclusion de la classe `java.awt.event.ActionEvent` ou de quelque autre classe ne se trouvant pas *directement* dans le paquetage `java.awt`;
- l'association entre structure de paquetages et structure de répertoires a pour effet de forcer les noms de paquetages à respecter les mêmes règles applicables à un nom de répertoire. Le nom sera significatif, évidemment; on y évitera les caractères accentués; et on cherchera à ne pas différencier deux paquetages par une simple différence de casse (ce qui, sous Windows, ne fonctionnerait pas).

Se dire « membre du paquetage `xyz` » implique d'insérer la commande `package xyz;` au début d'un fichier Java. En Java comme en C++, une classe est fermée pour ajout. Une fois atteint le lieu de l'accolade fermante de sa déclaration, on ne peut plus lui ajouter de membres.

En C#, les qualifications de sécurité usuelles (`public`, `protected` et `private`) rejoignent sémantiquement leurs homonymes en C++.

S'ajoutent à ces qualifications de sécurité la qualification **`internal`**, dont le sens *est accessible dans le même assemblage*, semblable à la qualification implicite *Package Private* de Java, et **`protected internal`**, une conjonction des qualifications `protected` et `internal` signifiant *accessible à la fois des dérivés de la classe et de l'assemblage dans lequel la classe réside*.

> Il est possible avec les langages `.NET` de définir des classes dites partielles, qui sont définies par morceaux dans plusieurs fichiers. Voir ***Classes partielles***, plus bas.

Une classe C# est toujours déclarée dans un espace nommé, et ne peut être `private` ou `protected`. Une classe peut donc être `public` (accessible de tous) ou `internal` (locale à l'assemblage). La qualification par défaut est `internal`, de manière analogue à la qualification implicite *privé au paquetage* de Java. Importer des noms d'un espace nommé s'y fait à l'aide du mot clé **`using`**. . Une classe peut être statique ou non

En `VB.NET`, les qualifications de sécurité `Public`, `Protected` et `Private` correspondent à celles en C++ et en C#. S'ajoutent les qualifications **`Friend`**, dont le sens correspond à celui de `internal` en C#, et **`Protected Friend`**, équivalent de `protected internal` en C# et de *Package Private* de Java.

Comme en C#, par défaut, les éléments d'un espace nommé sont `Friend`. Importer des noms d'un espace nommé s'y fait à l'aide du mot clé **`Imports`**. Une classe peut être partagée par son assemblage (`Shared`) ou non

Classes partielles

Les langages .NET comme C# et VB.NET supportent, depuis la version 2.0 de l'infrastructure, l'idée de ***classes partielles***. Cette mécanique pragmatique entre en conflit avec le principe ouvert/ fermé [POOv00], mais a suffisamment d'impacts positifs en pratique pour se mériter notre attention.

⇒ Une **classe partielle** est définie par morceaux dans plusieurs unités de traductions distinctes[42].

La syntaxe est simple : peu importe le langage, une classe partielle sera identifiée dans chaque fichier par le mot `partial` (ou `Partial` en VB.NET), indiquant clairement que tous les fichiers collaborant à sa définition sont conscients du fait qu'ils contribuent à une entité qui est essentiellement incomplète.

L'exemple proposé à droite comprend Z.X (car il peut y avoir des classes X dans d'autres espaces nommés, et elles ne sont pas nécessairement partielles) en deux parties et est en C# mais l'idée est la même en VB.NET et une classe partielle peut être faite d'un nombre arbitrairement grand de parties.

```
// fichier A.cs
namespace Z
{
   public partial class X
   {
      // ...une partie de Z.X
   }
}
// fichier B.cs
namespace Z
{
   public partial class X
   {
      // ...autre partie de Z.X
   }
}
```

Notez que le modèle .NET a ceci de particulier que certaines parties d'une classe partielle peuvent être écrites dans un langage alors que d'autres peuvent être rédigées dans un autre langage, du fait que tous les langages y utilisent le même modèle (et ont les mêmes limites).

Cette approche a un certain nombre de qualités. Entre autres, elle facilite le développement à plusieurs d'une classe donnée, permettant à des experts de différentes technologies de collaborer sur une même idée.

Elle permet aussi à un éditeur comprenant des générateurs de code[43] de générer, pour une classe donnée, une définition en plusieurs fichiers, montrant celui destiné à être modifié manuellement aux programmeuses et aux programmeurs tout en cachant celui qui devrait rester sous la responsabilité de l'éditeur.

En pratique, c'est une approche plus élégante que de placer des commentaires indiquant *Ne touchez pas à ce qui suit!* dans le code.

Évidemment, les classes partielles constituent une brèche de sécurité majeure. N'importe qui peut injecter n'importe quelle méthode dans une classe partielle et s'attribuer les privilèges de son choix de l'intérieur. Il faut toutefois que les divers fichiers implémentant chacun une partie d'une classe partielle donnée soient dans le même projet, ce qui réduit quelque peu les risques de dégâts ou de vandalisme.

---

[42] Voir http://msdn.microsoft.com/en-us/library/wa80x488(VS.80).aspx pour en savoir plus sur les détails techniques associés à ce mot clé.

[43] Comme *Visual Studio*, évidemment, puisqu'il s'avère que cet éditeur et l'infrastructure .NET sont deux produits de la même compagnie.

Méthodes d'extension

Même pour une classe fermée et qui n'est pas déclarée partielle, les langages .NET comme C# et VB.NET offrent, depuis la version 3.0 de l'infrastructure, un moyen d'enrichir son interface publique *apparente*. Ce moyen se nomme une méthode d'extension (en anglais : *Extension Method*). Comme toute chose, ces méthodes ont leurs avantages et leurs inconvénients.

L'idée derrière une méthode d'extension est de permettre au code client d'invoquer une méthode de classe (pas une méthode d'instance) de quelque classe que ce soit, et dont le premier paramètre est une référence sur une instance d'une classe X, comme s'il s'agissait d'une méthode d'instance de cette instance de X.

Par exemple, avec la syntaxe C#, on aurait l'exemple à droite. Notez que Test.Main() démontre deux syntaxes distinctes à l'invocation de Y.f(), l'une explicite (avec l'invocation Y.f(x), où x est une référence sur instance de X) et une autre implicite, qui illustre l'impact syntaxique des méthodes d'extension (invocation avec la syntaxe x.f()).

Le mot clé C# pour qualifier le premier paramètre d'une méthode d'extension est this, ce qui signifie *Pour l'invocation de cette méthode, considère que le premier paramètre jouera le rôle du paramètre implicite this d'une méthode d'instance*.

```
namespace Z
{
   class X
   {
      // ...
   }
   // ...
   static class Y
   {
      public static void f(this X x)
      {
         // ...
      }
   }
   class Test
   {
      public static void Main(string [] args)
      {
         X x = new X();
         Y.f(x); // Ok
         x.f();  // Ok aussi
      }
   }
}
```

Le langage VB.NET en arrive au même résultat, mais en ajoutant, juste avant la signature de la méthode d'extension, <System.Runtime.CompilerServices.Extension()> _ ce qui indique au compilateur que la méthode à la ligne suivante doit bénéficier d'un support syntaxique particulier.

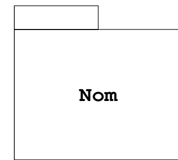
Les méthodes d'extension apparaissent comme un rêve pour programmeurs : avec elles, il est possible d'enrichir (en apparence) l'interface de toute classe d'une gamme arbitrairement riche et complexe de méthodes supplémentaires[44].

L'autre côté de la médaille est que cette pratique, quand elle implique un grand nombre d'individus, transforme tellement l'interface des classes qu'elle en vient souvent à transformer les langages eux-mêmes, ce qui rend le développement de systèmes complexes presque impossible. Il est donc sage d'utiliser les méthodes d'extension *lorsque cela s'avère nécessaire, point à la ligne*. Abuser des bonnes choses, en pratique, les rend indigestes.

[44] Cette pratique est ce qu'on nomme habituellement du *Monkey Patching*, et on la retrouve surtout dans les langages très dynamiques. Voir http://en.wikipedia.org/wiki/Monkey_patch pour des détails.

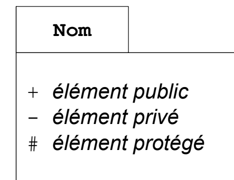
Regroupements et notation UML

Un regroupement logique comme un paquetage ou un assemblage se représente selon la notation UML proposée à droite, du moins pour qui souhait le présenter par son seul nom, sans porter un intérêt particulier aux entités qui en font partie.



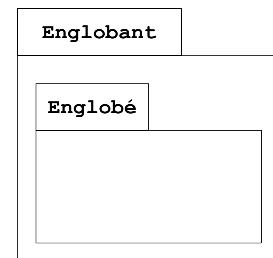
Cette forme peut être utile lorsque ce qui importe est de décrire le paquetage ou de présenter les relations de dépendance entre paquetages.

Dans le cas où l'intention est de montrer ce que contient un paquetage, on inscrira le nom du paquetage dans l'onglet qui le chapeaute.

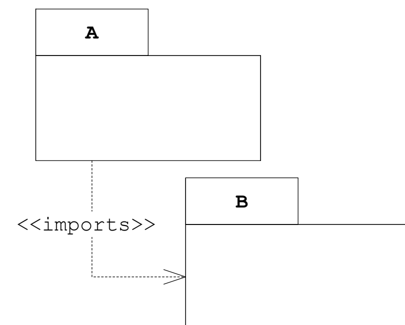


Les éléments d'un paquetage seront qualifiés comme privé, protégé ou public, comme le sont les membres d'un objet. La syntaxe pour discuter de *Package Private* et de ses synonymes n'est, à ma connaissance, pas standardisée.

Un paquetage pourra en contenir d'autres. La notation proposée à droite sera alors utilisée. Dans ce schéma, de manière visible, le paquetage Englobant contient le paquetage Englobé.



Notez qu'il est possible que certaines fonctionnalités représentables sous forme UML n'aient pas d'équivalent dans un langage ou l'autre. Par exemple, une fonction globale à un espace nommé est possible en C++ mais ne l'est pas en Java, C# ou VB.NET.



Enfin, il est possible de décrire des relations de dépendance entre paquetages. Plusieurs relations de dépendance sont possibles, la plus simple étant l'utilisation par un paquetage d'éléments dans un autre paquetage.

Dans le schéma à droite, le paquetage A importe des éléments de B, ce qui implique que A dépend de B.

Note technique sur `class` et `struct`

Grossièrement, il est raisonnable de prétendre que la POO exprime des idées en termes d'objets, donc à partir de classes, d'instances et de relations entre elles. L'idée de classe y est centrale, ce qui explique que le mot class apparaisse dans autant d'exemples et d'illustrations.

Il se trouve qu'en C++ (mais pas nécessairement dans d'autres langages), le mot `struct` a un sens connexe au mot `class`, ce qui fait que nous utiliserons parfois l'un, parfois l'autre dans ce document. Les quelques subtilités auxquelles il faudra faire attention sont résumées dans le tableau suivant.

class	**struct**
Les membres sont **privés** jusqu'à preuve du contraire.	Les membres sont **publics** jusqu'à preuve du contraire.
L'héritage est **privé** jusqu'à preuve du contraire.	L'héritage est **public** jusqu'à preuve du contraire.
Conceptuellement, constitue surtout une unité de **sécurité**.	Conceptuellement, constitue surtout une unité de **regroupement**.

En fait, les deux idées sont, en C++, équivalentes, à des nuances techniques pointues près. Les deux peuvent avoir des membres privés, protégés et publics, des constructeurs, des destructeurs, des attributs, des méthodes, *etc*. Pour cette raison, quand nous examinerons des classes simples où tous les membres sont publics, nous utiliserons parfois des `struct` plutôt que des `class`, strictement pour simplifier le propos et éviter de la redondance.

Par exemple, la `class` nommée `X` et la `struct` nommée `X`, ci-dessous, sont identiques. Puisque la `struct` n'a que des membres publics, nous pourrions choisir la privilégier dans nos exemples si cela n'affecte pas le propos.

```
class X {
public:
   int f() const noexcept {
      return 3;
   }
};
```

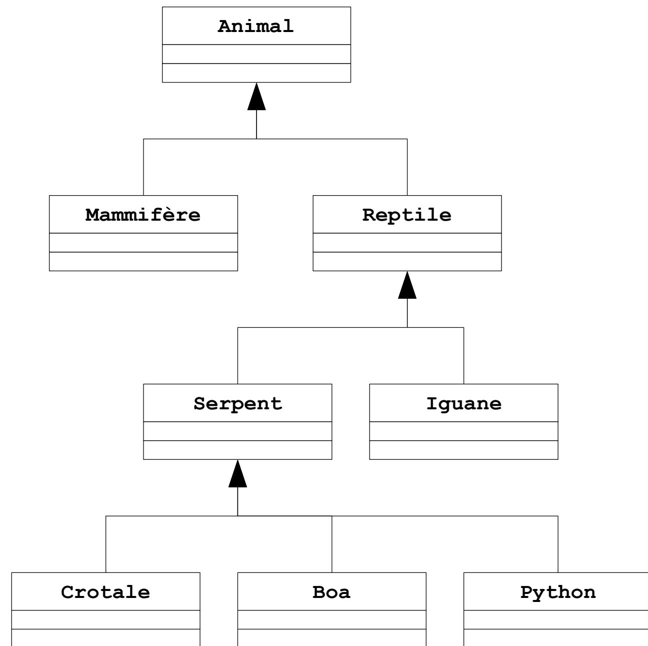
```
struct X {
   int f() const noexcept {
      return 3;
   }
};
```

Le choix d'utiliser `struct` pour les classes dont les membres sont tous publics et pour lesquelles le propos ne relève pas de l'encapsulation rejoint les usages acceptés. Être en contact avec cette écriture ne peut donc pas nuire. Évidemment, la classe demeure l'unité à privilégier dans la majorité des cas.

Exercices – Série 02

Suivent, des exercices portant sur la notion d'héritage, dans des situations où elle peut s'avérer (ou non) une bonne idée.

EX00 – Présumant l'arborescence de classes suivantes, où on lira chaque flèche du bas vers le haut comme *hérite publiquement de.*



Présumant aussi les prototypes de sous-programmes suivants :

```
void AnalyseSerpentine(Serpent&);
void AnalyseAnimale(Animal&);
void AnalyseReptilienne(Reptile&);
```

Donnez la liste des classes dont on peut en toute légalité passer une instance en paramètre…

…à la procédure `AnalyseSerpentine()` :______________________________

____

____

…à la procédure `AnalyseAnimale()` : ______________________________

____

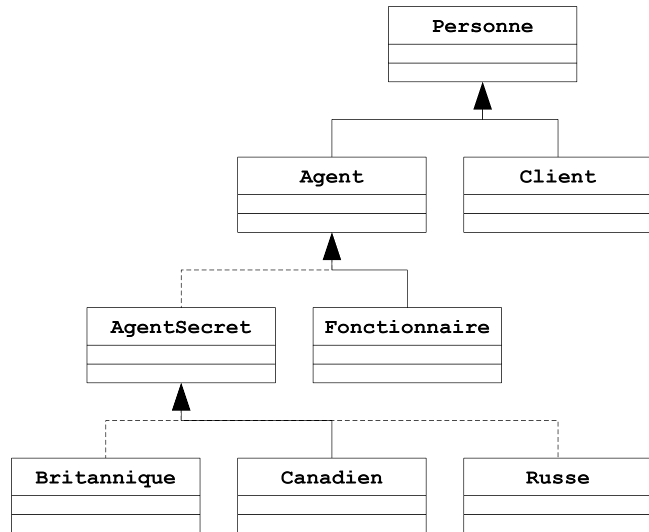
____

…à la procédure `AnalyseReptilienne()` : ______________________________

____

____

EX01 – Présumant l'arborescence de classes suivantes, où on lira chaque flèche du bas vers le haut comme *hérite publiquement de* et chaque flèche pointillée du bas vers le haut comme *hérite de manière privée de*.



Présumant aussi les prototypes de sous-programmes suivants :

```
void p0(Personne&);
void p1(Agent&);
void p2(AgentSecret&);
```

Donnez la liste des classes dont on peut en toute légalité passer une instance en paramètre…

…à la procédure `p0()` :______________________________

____

____

…à la procédure `p1()` :______________________________

____

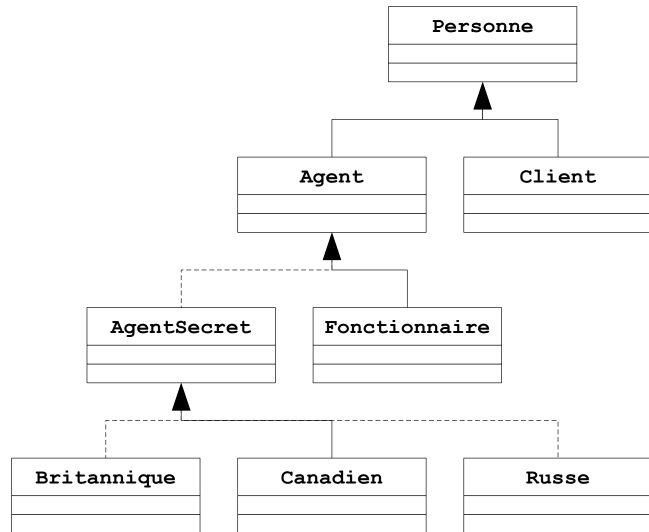
____

…à la procédure `p2()` :______________________________

____

____

EX02 – Présumant l'arborescence de classes suivantes, où on lira chaque flèche du bas vers le haut comme *hérite publiquement de* et chaque flèche pointillée du bas vers le haut comme *hérite de manière privée de*.



Si vous travaillez à une méthode retournant une Personne terminant par :

```
return *this;
```

alors à quelle(s) classes cette méthode peut-elle appartenir? ______________________________

____

____

Si vous travaillez à une méthode retournant un Agent et se terminant par :

```
return *this;
```

alors à quelle(s) classe(s) cette méthode peut-elle appartenir? ______________________________

____

____

Si vous travaillez à une méthode retournant un AgentSecret et se terminant par :

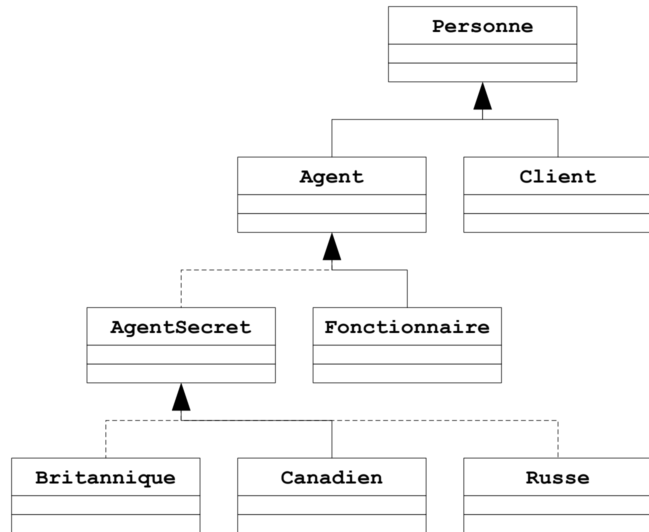
```
return *this;
```

alors à quelle(s) classe(s) cette méthode peut-elle appartenir? ______________________________

____

____

EX03 – Présumant l'arborescence de classes suivantes, où on lira chaque flèche du bas vers le haut comme *hérite publiquement de* et chaque flèche pointillée du bas vers le haut comme *hérite de manière protégée de*.



Si vous travaillez à une méthode retournant une `Personne` terminant par :

```
return *this;
```

alors à quelle(s) classes cette méthode peut-elle appartenir? ______________________________

____

____

Si vous travaillez à une méthode retournant un `Agent` et se terminant par :

```
return *this;
```

alors à quelle(s) classe(s) cette méthode peut-elle appartenir? ______________________________

____

____

Si vous travaillez à une méthode retournant un `AgentSecret` et se terminant par :

```
return *this;
```

alors à quelle(s) classe(s) cette méthode peut-elle appartenir? ______________________________

____

____

EX04 – Cet exercice se veut plus intégrateur, et demande donc plus de travail que les précédents. Imaginez-vous œuvrant chez un constructeur automobile célèbre, **AUTOCO**, où l'on modélise les voitures avant de commander leur construction.

La compagnie **AUTOCO** fabrique plusieurs types de véhicules: des motocyclettes, des véhicules de type « tout-terrain », des automobiles sport, des berlines plus conventionnelles[45]…

Élaborez une structure de classes par héritage et par composition qui permette de représenter l'éventail de produits de la compagnie. On veut :

- qu'il soit possible de traiter tout véhicule en tant que véhicule;
- qu'il soit possible de traiter toute automobile en tant qu'automobile;
- que chaque véhicule soit fait de ses différentes pièces principales (pas besoin d'aller dans le grand détail, mais vous pouvez présumer l'existence de classes décentes pour la plupart des composants communs, par exemple des roues ou des essuie-glaces), de manière à ce qu'on puisse les changer au besoin;
- on ne veut pas nécessairement une classe par véhicule possible (ce serait abusif). L=esa classes forment des familles, et il faut à un moment donné les instancier.

> Il y a beaucoup, beaucoup de manières d'arriver à un résultat convenable ici, surtout avec des spécifications aussi vagues. L'objectif de l'exercice est de réfléchir à la place des attributs, méthodes et de l'héritage dans une structure plus riche.

Réflexion

Vous avez en tête plusieurs fournisseurs potentiels pour une même bibliothèque, avec les mêmes déclarations de fonctions et classes dans chaque cas; vous hésitez entre les différentes offres mais devez investir dans l'une ou l'autre.

Expliquez comment l'emploi d'espaces nommés par vos fournisseurs peut vous faciliter la tâche pour ce qui est d'organiser une batterie de tests en vue de choisir le produit d'un d'entre eux plutôt que celui d'un autre.

[45] Prenez un catalogue d'un fabricant connu si vous êtes à la recherche d'une liste exhaustive.

Polymorphisme

Pourquoi cette section?

Le polymorphisme est l'un de ces éléments clés qui rendent vraiment l'approche OO indispensable pour le développement de projets gros et complexes. Permettant de généraliser et de spécialiser sérieusement et fonctionnellement les objets, cette mécanique donne aux logiciels conçus de manière OO une puissance qui les détache du lot.

Nous centrerons nos efforts sur ce qu'il y a lieu d'appeler le polymorphisme dynamique (*Late Binding*, en anglais), mais le mot « polymorphisme » est quelque peu galvaudé au moment d'écrire ces lignes et peut prendre plusieurs autres sens selon le contexte.

Maintenant que nous avons examiné ce qu'est, à la base, l'*héritage*, et que nous comprenons mieux l'impact de dériver une classe d'une autre classe à l'aide de l'une ou l'autre des spécifications d'accès à notre disposition :

- nous réexaminerons la ***surcharge de méthodes***. Dans ce cas-ci, nous serons intéressés à ce qui se produit lorsqu'une classe enfant écrit une méthode qui porte *exactement* la même signature qu'une méthode de son parent;
- ceci nous permettra d'introduire l'idée de **polymorphisme**, avec un nouveau mot clé (le mot `virtual`). Cette idée deviendra pour nous l'un des plus importants, des plus indispensables pivots de la pensée objet; puis
- nous examinerons plus en détail **comment** (et selon quelles contraintes) un enfant peut **faire un usage direct et explicite des membres d'un parent**.

Retour sur la surcharge de méthodes

Imaginons la situation suivante. Nous avons une classe `Personne` assez simple qui ressemble à peu près à ceci :

Personne.h	Personne.cpp

`#ifndef PERSONNE_H`
`#define PERSONNE_H`
`#include <string>`
`class Personne {`
`   std::string nom_;`
`public:`
`   std::string nom() const;`
`   Personne(const std::string &);`
`   void qui_suis_je() const;`
`};`
`#endif`
	`#include "Personne.h"`
	`#include <string>`
	`#include <iostream>`
	`using namespace std;`
	`string Personne::nom() const {`
	`   return nom_;`
	`}`
	`Personne::Personne(const string &nom) : nom_{nom} {`
	`}`
	`void Personne::qui_suis_je() const {`
	`   cout << "Je suis " << nom() << endl;`
	`}`

Imaginons aussi que nous avons les classes `Jim`, `Joe` et `Bob` telles que :

Jim.h

```
#ifndef JIM_H
#define JIM_H
#include "Personne.h"
#include <string>
class Jim : public Personne {
public:
   Jim(const std::string&);
   ~Jim() = default;
   void qui_suis_je() const;
};
#endif
```

Joe.h

```
#ifndef JOE_H
#define JOE_H
#include "Personne.h"
#include <string>
class Joe : public Personne {
public:
   Joe(const std::string&);
   ~Joe()= default;
   void qui_suis_je() const;
};
#endif
```

Bob.h

```
#ifndef BOB_H
#define BOB_H
#include "Personne.h"
#include <string>
class Bob : public Personne {
public:
   Bob(const std::string&);
   ~Bob() = default;
   void qui_suis_je() const;
};
#endif
```

Jim.cpp

```
#include "Jim.h"
#include <iostream>
#include <string>
using namwspace std;
Jim::Jim(const string &nom)
   : Personne{nom + ", Jim"}
{
}
void Jim::qui_suis_je() const {
   cout << "Mon nom est "
        << nom() << endl;
}
```

Joe.cpp

```
#include "Joe.h"
#include <iostream>
#include <string>
using namespace std;
Joe::Joe(const string &nom)
   : Personne{nom + ", Joe"}
{
}
void Joe::qui_suis_je() const {
   cout << "J'suis, euh, "
        << nom() << endl;
}
```

Bob.cpp

```
#include "Bob.h"
#include <iostream>
#include <string>
using namespace std;
Bob::Bob(const string &nom)
   : Personne{nom + ", Bob"}
{
}
void Bob::qui_suis_je() const {
   cout << "Coucou, je suis "
        << nom() << endl;
}
```

Tout ceci devrait vous sembler légal. Maintenant, apposons notre regard sur les programmes qui suivent, et essayons de prédire le résultat de leur exécution.

En exécutant le programme console ci-dessous, qu'obtiendra-t-on à l'écran?

Programme 0

```
#include "Jim.h"
#include "Joe.h"
#include "Bob.h"
int main() {
   Jim jim{"Tremblay"};
   Joe joe{"Tremblay"};
   Bob bob{"Tremblay"};
   jim.qui_suis_je();
   joe.qui_suis_je();
   bob.qui_suis_je();
}
```

Réponse à la page suivante…

Vous l'avez sans doute deviné, ce qui sera affiché sera :

```
Mon nom est Tremblay, Jim
J'suis, euh, Tremblay, Joe
Coucou, je suis Tremblay, Bob
```

Ce qui se tient, d'ailleurs. Vous êtes invité(e)s à suivre pas à pas les constructeurs et les méthodes `qui_suis_je()` plus haut pour vous en convaincre.

Petit détail : si `Joe` avait voulu appeler explicitement le `qui_suis_je()` de `Personne`, par exemple pour afficher `Vous ne me connaissez pas, mais... Je suis Tremblay, Joe`, il aurait pu le faire comme ceci :

```
void Joe::qui_suis_je() const {
   cout << "Vous ne me connaissez pas, mais ";
   Personne::qui_suis_je();
}
```

On se rappellera à ce sujet l'exemple avec les classes `Pikachu` et `Pichu` et la méthode `bzzz()`, dans la section ***Héritage privé et membres du parent***, plus haut.

Un peu plus amusant, maintenant. En exécutant le programme console ci-dessous, qu'obtiendra-t-on à l'écran?

Programme 1

```
#include "Jim.h"
#include "Joe.h"
#include "Bob.h"
int main() {
   Jim *jim = new Jim{"Tremblay"};
   Joe *joe = new Joe{"Tremblay"};
   Bob *bob = new Bob{"Tremblay"};
   jim->qui_suis_je();
   joe->qui_suis_je();
   bob->qui_suis_je();
   delete jim;
   delete joe;
   delete bob;
}
```

Réponse à la page suivante…

Encore une fois, ce qui sera affiché sera :

```
Mon nom est Tremblay, Jim
J'suis, euh, Tremblay, Joe
Coucou, je suis Tremblay, Bob
```

La même logique s’applique, et vous avez sans doute déjà compris la mécanique derrière ce résultat. Un peu plus amusant, maintenant. En exécutant le programme console ci-dessous, qu’obtiendra-t-on à l’écran?

Programme 2

```
#include "Personne.h"
#include "Jim.h"
#include "Joe.h"
#include "Bob.h"
int main() {
   const int NB_PERSONNES = 3;
   Personne *p[NB_PERSONNES];
   p[0] = new Jim{"Tremblay"}; // Légal : un Jim est une Personne
   p[1] = new Joe{"Tremblay"}; // Légal : un Joe est une Personne
   p[2] = new Bob{"Tremblay"}; // Légal : un Bob est une Personne
   for (int i = 0; i < NB_PERSONNES; i++) {
      p[i]->qui_suis_je();
      delete p[i];
   }
}
```

Réponse à la page suivante…

Cette fois, ce qui vous surprendra peut-être, ce qui sera affiché sera :

```
Je suis Tremblay, Jim
Je suis Tremblay, Joe
Je suis Tremblay, Bob
```

Or, il est probable qu'il ne s'agisse pas là de ce que nous aurions souhaité.

Pour examiner le même problème avec une mise en situation plus concrète, remplaçons `Jim`, `Joe` et `Bob` par un bouton de commande, une animation `3D` et une liste déroulante. De même, remplaçons la classe `Personne` par la classe `ComposantAffichable`, et la méthode `qui_suis_je()` par une autre nommée `afficher()`.

On devrait alors comprendre ce vers quoi nous nous dirigeons ici : il serait souhaitable que demander à un composant affichable *quel qu'il soit* de s'afficher fasse en sorte que ce composant s'affiche le mieux possible *selon ses besoins et sa fonction propre*. On ne veut pas voir s'afficher un composant générique, après tout. De la même manière, on veut ici que `Jim` se présente *comme un* `Jim`, pas comme une simple `Personne` générique.

Introduction au polymorphisme

Le polymorphisme est l'une des idées plus importantes (et les plus mal comprises) du modèle OO tout entier. Plusieurs gens se voulant des gourous du sujet n'ont pas la moindre idée de ce qu'il veut réellement dire. *Méfiez-vous des fraudeurs!*

La problématique présentée par notre mise en situation tient au fait que nous appelons chaque fois la méthode `qui_suis_je()` à travers une indirection sur une `Personne` (ici, à travers une `Personne*`, mais la situation est identique à travers des références). Ceci sollicite la méthode `qui_suis_je()` définie *pour la classe* `Personne`.

On pourrait croire qu'il serait utile qu'à l'exécution, *la méthode* `qui_suis_je()` *la plus exacte possible pour l'instance vers laquelle on pointe* soit appelée. Cela aurait l'avantage de permettre de rassembler, à travers des pointeurs du type de leur parent commun, des instances de classes dérivées de ce parent, et de traiter chacune comme un cas particulier de ce parent, tout en obtenant le meilleur support fonctionnel possible[46].

Nous ne voulons pas écrire un immense sous-programme examinant le type et les données d'un objet puis décidant, de manière autoritaire, d'une marche à suivre; nous voulons une réaction subjective et personnelle à l'invocation de la méthode : nous voulons du **polymorphisme**[47].

Bref aperçu

Le cas de `qui_suis_je()` est un cas banal mais courant de polymorphisme. Lorsque nous rencontrons des gens, nous entrons en interaction avec eux et nous avons des attentes protocolaires quant à ces interactions. Saluer quelqu'un implique habituellement une réaction cordiale, mais cette réaction variera selon les individus.

Structurellement, nous interagissons avec des humains sur la base qu'ils sont, effectivement, des humains (une classe). Nos attentes protocolaires sont basées sur cette réalité (une interface publique). Sur le plan comportemental, nous estimons normal que chaque humain réagisse à sa manière, et nous serions les premiers étonnés de voir tous les humains réagir de manière absolument identique (mêmes mots, même ton, même attitude) à une salutation donnée.

Cette personnalisation de la réponse à une invocation de méthode, cette manière subjective qu'a chaque classe, parfois même chaque instance, de réagir à sa manière, est l'essence-même du polymorphisme. La mécanique des langages OO transforme cette précieuse réalité de la vie courante en réalité des systèmes informatiques.

---

[46] Présumant, bien sûr, que l'effet voulu soit effectivement d'avoir accès à la version la plus spécialisée possible des méthodes appelées. Si vous voulez l'effet générique, alors le polymorphisme n'est pas pour vous, du moins dans le cas en question.

[47] Du grec pour *plusieurs formes*. Attention : bien des gens savent ce détail, mais ne comprennent pas *ce qu'est* le polymorphisme… Si un individu n'a pas compris le polymorphisme, cet individu n'a pas non plus compris l'approche OO. Vous disposez ici d'un bon test pour valider la qualité d'un prospect dans une équipe de développement : celles et ceux qui ne connaissent de l'approche que ce qu'ils ont lu dans un livre sont souvent tombés sur un auteur n'ayant pas, lui non plus, compris le polymorphisme… avec les conséquences que vous pouvez d'ores et déjà imaginer.

Exemple concret

Bien que le comportement polymorphique ne soit pas le comportement par défaut d'une méthode en C++, on peut l'implanter sans peine à l'aide du mot clé **`virtual`**. Voici comment :

- on choisira d'abord la classe qui servira de ***racine polymorphique*** pour le comportement visé. Dans notre cas, cette classe sera `Personne` car (comme le montre notre programme de test) c'est à partir d'indirections sur `Personne` que nous souhaitons travailler;
- la racine polymorphique sera une classe générale, qui représentera au moins le niveau d'abstraction à partir du duquel nous souhaitons construire nos programmes. Ici, notre programme demandera à des personnes de se présenter, ce qui explique le choix de `Personne` à titre de racine;
- on identifiera ensuite les comportements pour lesquels le polymorphisme est souhaité. Ici, une seule méthode doit être polymorphique, soit la méthode `qui_suis_je()`, mais de manière générale, il pourrait y en avoir un nombre arbitrairement élevé;
- chaque méthode polymorphique sera déclarée, *au moins là où elle est spécifiée la première fois* (ici, dans la déclaration de la classe `Personne`), comme étant *virtuelle* (voir à droite); et c'est tout.

```
#ifndef PERSONNE_H
#define PERSONNE_H
#include <string>
class Personne {
   std::string nom_;
public:
   std::string nom() const;
   Personne(const std::string&);
   virtual ~Personne() = default;
   virtual void qui_suis_je() const;
};
#endif
```

> Pour des raisons qui seront expliquées sous peu, une classe vouée à un usage polymorphique devrait toujours avoir un destructeur virtuel, ce qui explique que la qualification `virtual`, dans notre exemple, a été apposée à la fois à la méthode `Personne::qui_suis_je()` et au destructeur, même si le destructeur utilisé ici est trivial.

Le simple ajout du mot clé **`virtual`** à la déclaration (et seulement à la déclaration) de la méthode `qui_suis_je()` de la classe `Personne` signifie ce qui suit.

⇒ Si, à travers une indirection (pointeur ou référence) vers une `Personne` ou l'une de ses classes dérivées, on invoque la méthode `qui_suis_je()`, alors plutôt que de prendre la méthode `qui_suis_je()` du type *immédiatement* pointé, on cherchera ***à l'exécution*** la méthode la plus spécialisée pour l'objet *réellement* pointé.

Polymorphisme et indirections

Le polymorphisme ne s'applique, en C++, que sur des instances utilisées de manière *indirecte*, donc à travers des pointeurs ou des références. Cela tombe sous le sens, quand on y pense. Le polymorphisme, concrètement, permet d'invoquer, à travers une abstraction plus élevée (quelque chose de plus général), des méthodes plus spécialisées, d'une classe plus spécifique.

La classe plus abstraite est donc nécessairement un ancêtre, et choisit à même son design de définir certains de ses comportements comme susceptibles d'être spécialisés par ses enfants. Dans notre exemple, toute `Personne` sait se présenter de manière générale mais les dérivés de `Personne` peuvent se présenter différemment des instances de `Personne` en général s'ils le jugent opportun. *Le polymorphisme permet aux enfants de spécialiser un comportement d'un parent mais ne les y oblige pas*.

On voudra donc, en pratique, traiter plusieurs cas spécifiques (des `Jim`, des `Joe`, des `Bob` et peut-être même d'autres sortes de `Personne`) sur un plan plus abstrait, à partir d'une idée plus générale (ces objets sont, au fond, des instances de `Personne` et c'est sur cette base que nous voulons travailler avec eux).

On manipulera en apparence des instances de `Personne`, mais ce sera en général des instances de classes dérivées que nous manipulerons. Nous passerons donc vraisemblablement par des indirections.

Prenons un autre exemple, classique (et déjà quelque peu exploré dans [POOv00]) : les formes géométriques. Imaginons que, selon notre design, toute forme puisse se déplacer. Imaginons aussi que le déplacement d'une forme s'accompagne de divers effets visuels. Les extraits de code ci-dessous montrent un cas polymorphique (passant par une indirection) et un cas qui ne l'est pas (reposant sur une sémantique d'accès direct).

```
class Forme {
   // ...
public:
   virtual void deplacer();
   virtual ~Forme();
};
class Rectangle : public Forme {
   // ...
public:
   virtual void deplacer();
};
```

Le polymorphisme est possible	Le polymorphisme n'a pas de sens

`Forme *pf = new Rectangle;`
`// pf pointe vers une Forme ou vers`
`// n'importe quel dérivé public de Forme`
`// ...`
`pf->deplacer();`
`// Invoque le Deplacer() de l'instance pointée`
`// (polymorphisme!) dans la mesure où la méthode`
`// Forme::Deplacer() est qualifiée virtual`
	`Forme f;`
	`// f est nécessairement une Forme`
	`// ...`
	`f.deplacer(); // Forme::deplacer()`

Autre démonstration du jeu entre polymorphisme et indirections : l'exemple à droite, où `f()` reçoit en paramètre une `Personne` par copie et où `g()` reçoit en paramètre une `Personne` par référence (constante, mais ce détail est sans importance pour notre discours).

Le programme principal crée une `Personne`, l'instance nommée `p`, et un `Joe`, l'instance nommée `j`, puis invoque successivement `f(p)` et `f(j)`.

Ces invocations passant à `f()` une copie de la `Personne` originale, `f()` travaillera sur la copie locale mise à sa disposition lors de l'invocation. Dans les deux cas, le type du paramètre sera `Personne`, et la méthode appelée lorsque `f()` sollicitera `p.qui_suis_je()` sera `Personne::qui_suis_je()`.

Par la suite, le programme principal invoque `g(p)` et `g(j)`.

Ces invocations passant à `g()` une référence vers l'original. Puisque cette référence mène vers une `const Personne`, seules des méthodes déclarées au niveau de `Personne` (et qualifiées `const`, par la force des choses) seront disponibles à la compilation.

Cependant, la version de la méthode invoquée lorsque `g()` sollicitera `p.qui_suis_je()` dépendra du type vers lequel mène réellement la référence[48]. Dans un cas, ce sera la version de `Personne`, et dans l'autre ce sera la version de `Joe`.

```
#include "Personne.h"
#include "Joe.h"
void f(Personne);
void g(const Personne&);
int main() {
   Personne p{"NGuyen"};
   Joe j{"Nguyen"};
   f(p);
   f(j);
   g(p);
   g(j);
}
void f(Personne p) {
   p.qui_suis_je();
}
void g(const Personne &p) {
   p.qui_suis_je();
}
```

[48] Le type `Personne` signifie `Personne`, mais le type `Personne&` signifie *au moins* `Personne`.

Autre exemple, à partir d'une stratégie moins répandue mais tout aussi légale, l'extrait ci-dessous traitera la référence à un `x` (qui mène, en fait, à un `y`) comme un `y` lorsqu'on sollicitera sa méthode virtuelle `Yo()`.

En effet, dans `main()`, le symbole `x0` est une référence sur un `X`, donc une indirection vers au moins un `X`. Le fait que `x0` soit lié à `y` (une instance de `Y`), ce qui est légal puisque `Y` est un dérivé public de `X`, a pour conséquence de faire de `x0` une entité exposant les méthodes de `X` mais permettant, dans le cas des méthodes virtuelles, d'invoquer la version définie dans `Y` de ces méthodes.

L'invocation de `x0.Yo()` dans `main()` est une invocation polymorphique parce que `x0` est une indirection. Conséquemment, cette invocation sollicitera la méthode `Y::Yo()`.

Par contre, l'invocation `x1.Yo()`, aussi réalisée dans `main()`, n'est pas une invocation polymorphique car `x1` est un `X`, pas une indirection vers quelque chose qui soit au moins un `X`. Cette invocation sollicitera donc la méthode `X::Yo()`.

```
#include <iostream>
using namespace std;
struct X {
   virtual void Yo(ostream &os) const {
      os << "Yo de X" << endl;
   }
};
struct Y : X {
   void Yo(ostream &os) const {
      os << "Yo de Y" << endl;
   }
};
int main() {
   Y y;
   X &x0 = y;
   x0.Yo(cout);
   X x1;
   x1.Yo(cout);
}
```

Réflexion 01.0 : Vous avez peut-être remarqué que `X::Yo()` et `Y::Yo()` auraient, en temps normal, pu être des méthodes de classes puisqu'elles n'accèdent, pour compléter leur tâche, à aucun membre d'instance. Pourtant, ici, ce sont des méthodes d'instance. À votre avis, quelle est la raison derrière cette décision technique? (voir ***Réflexion 01.0 : le polymorphisme est subjectif*** pour une discussion).

Relevez la différence entre le volet statique et le volet dynamique du polymorphisme :

- à la compilation (volet statique), le compilateur s'assure que seules les opérations permises sur les types impliqués sont entreprises par le programme (ici : on ne peut invoquer à partir d'une instance `Personne` que des méthodes disponibles à partir de la classe `Personne`, que cette instance soit accédée directement ou non); alors que
- à l'exécution (volet dynamique), la mécanique découvrira la bonne version de chaque méthode en fonction des types réellement impliqués dans les opérations.

L'un des aspects techniques qui rend le polymorphisme naturel en Java et dans les langages `.NET` est que l'accès direct à un objet y est impossible: tous les accès à un objet s'y font nécessairement à travers une référence. Voir [POOv00], ***appendice 00*** pour plus de détails..

Cette combinaison faite de la protection statique offerte par le compilateur et de la souplesse dynamique résultant de la découverte à l'exécution des types réellement impliqués dans les opérations polymorphiques sur des indirections est d'une valeur inestimable.

Polymorphisme et surcharge

Il semble y avoir une confusion qui règne à l'occasion quant à la nature du ***polymorphisme***, et à sa relation avec l'idée de ***surcharge***. Les deux sont pourtant bien différents.

L'idée de **surcharge**[49] signifie la possibilité d'avoir plusieurs sous-programmes portant le même nom. Plusieurs formes, soit, mais une mécanique strictement statique.

Au premier niveau, on peut voir apparaître plusieurs sous-programmes portant le même nom, mais avec des signatures de paramètres différentes. En partie dû au besoin d'avoir plus d'un constructeur, le support à la surcharge de sous-programmes est nécessaire dans la majorité des langages OO.

```
// surcharge: même nom,
// signatures différentes
int f();
int f(int);
int f(float);
```

À un autre niveau, on trouve la possibilité pour une classe enfant de remplacer une méthode de l'un de ses parents. Selon la manière par laquelle on procédera, ceci mènera parfois vers l'emploi de polymorphisme, et cachera parfois simplement la méthode du parent.

Il y a plusieurs variantes possibles ici, chacune d'elles avec ses petites nuances. Il faut être prudent(e) quant aux glissements sémantiques possibles.

Pour réaliser la surcharge d'un sous-programme, le compilateur doit trouver d'autres moyens de distinguer deux sous-programmes que leur nom. Tel que nous l'avons vu précédemment, cela signifie pour C++ que ce qui définit réellement le nom d'un sous-programme est son nom, le nombre de paramètres qu'il accepte et le type de ces paramètres (en plus, pour les méthodes d'instance, des qualifications `const` et `volatile`).

```
struct X {
   int f() const;
};
struct Y : X {
   // dissimule X::f() pour un Y
   int f() const;
};
```

[49] En anglais, on verra deux sens différents du terme avec *Overloading* et *Overriding*; le second est plus près du polymorphisme que le premier. La littérature récente parle parfois de *Run-Time Binding* pour mettre en relief le caractère foncièrement dynamique du polymorphisme.

Règle générale, le nom d'un sous-programme une fois compilés (et disponible pour l'édition des liens) est composé de son nom apparent et d'un amalgame de ses paramètres et de leur type[50].

La surcharge est donc une mécanique ***statique***, au sens où elle entre en considération à la compilation seulement, et s'applique (dans les langages supportant à la fois l'approche structurée et l'approche OO) à la fois aux sous-programmes globaux et aux méthodes d'instances. Le polymorphisme est une mécanique ***dynamique***[51]. De par le polymorphisme, la mécanique d'un langage OO ira trouver à l'exécution, pour une méthode polymorphique à travers un pointeur ou une référence vers une instance d'une classe donnée, la version la plus spécialisée qui soit d'une méthode ayant la même signature que la méthode appelée.

```
struct B {
   virtual int f() const;
   virtual ~B() = default;
};
struct D : B {
   int f() const;
};
int main() {
   B b;
   D d;
   B &r = d;
   b.f(); // B::f()
   d.f(); // D::f()
   r.f(); // D::f()
}
```

Mais le polymorphisme statique …?

Il existe des techniques de programmation générique [POOv02] permettant de réaliser ce qu'on nomme du polymorphisme statique. Ces techniques dépassent toutefois *de loin* la simple surcharge de sous-programmes. Nous reviendrons sur ce sujet en temps et lieu.

[50] L'opération servant à générer ce nom est bien sûr réversible. Si un programme C++ et un programme C (ou Fortran, ou écrit dans un autre langage pour lequel une différenciation de sous-programmes par le nom seulement suffit) doivent communiquer, on aura recours à des fonctions à nom unique pour tracer le lien entre eux. Vous pouvez consulter l'aide en ligne pour la rubrique `extern "C"` si vous voulez en savoir plus.

[51] En apparence, du moins. Pour des fins d'efficacité, certains langages (comme C++, on s'en doute) préparent le terrain pour faciliter des appels de méthodes plus rapides dès la compilation des classes.

Polymorphisme et contrats

Imaginons les classes `X` et `Y`, à droite. La classe `X` déclare une méthode non polymorphique, `f()`, et une méthode virtuelle, `g()`. Comme toute bonne classe polymorphique, évidemment, `X` déclare aussi son destructeur comme étant virtuel. La classe `Y`, quant à elle, spécialise `g()` et expose aussi une nouvelle méthode, `h()`.

Le programme principal qui suit ces déclarations illustre, à partir de ces classes, les implications des contrats polymorphiques :

```
struct X {
   void f();
   virtual void g();
   virtual ~X();
};
struct Y : X {
   void g();
   void h();
};
int main() {
   X x;
   Y y;
   X &r = y;
   x.f(); // Ok
   x.g(); // X::g()
   y.f(); // Ok, hérité de X
   y.g(); // Y::g(), sans polymorphisme
   y.h(); // Ok
   r.f(); // Ok
   r.g(); // Y::g(), par polymorphisme
   r.h(); // Illégal (pas déclaré dans X)
}
```

- à travers `x`, de type `X`, seules les méthodes exposées dans `X` sont disponibles, et aucun polymorphisme ne se produit (`x` est un objet, pas une indirection);
- à travers `y`, de type `Y`, seules les méthodes exposées dans `Y` sont disponibles, incluant celles publiques et héritées du parent `X`. Aucun polymorphisme ne s'applique (`y` n'est pas une indirection), mais `Y::g()` est préféré à `X::g()` puisque `y` est d'abord un `Y`;
- enfin, `r` est une indirection de type `X&` vers un `Y`. Seules les méthodes déclarées dans `X` y sont donc accessibles : on ne peut écrire `r.h()`, même si l'objet référé possède cette méthode, car la méthode `h()` ne fait pas partie de l'interface publique de `X`;
- par contre, `r` étant une indirection plutôt qu'un objet, le polymorphisme s'applique à travers lui. Ainsi, `r.g()` invoque `Y::g()` du fait que `r` réfère, en fait, à un `Y`.

Dans un design comme celui proposé dans cet exemple, les conceptrices et les concepteurs des classes auraient voulu exprimer qu'il y a des raisons d'utiliser plusieurs sortes de `X` (qu'on parle de `X`, de `Y` ou d'autres descendants de `X`) sur la base que ce sont tous au moins des `X`, en particulier à travers le service polymorphique `g()` qu'ils ont tous en commun, mais qu'il y aura des programmes qui utiliseront des `Y` en tant que `Y` (du fait que `Y` expose certaines méthodes publiques qui lui sont propres, par exemple `h()`).

Domaines d'application

Dans quel genre de situation, généralement, appliquera-t-on le polymorphisme? S'agit-il d'une mécanique qu'on peut utiliser dans n'importe quelle circonstance? Question connexe, mais différente : est-ce en tout temps une mécanique utile? Toutes les méthodes ne devraient-elles pas être polymorphiques?

En pratique, en fait, la réponse est non, ce qui peut en surprendre plusieurs. Certains estiment que le comportement d'une méthode devrait être polymorphique par défaut : en Java, par exemple, outre les constructeurs, les méthodes d'instance sont *polymorphiques jusqu'à preuve du contraire*. Ce biais, compréhensible dans certaines écoles de pensée, est injustifié en pratique dans un système de types complet.

Nous couvrirons tout d'abord quelques cas pour lesquels le polymorphisme n'est pas indiqué, puis nous examinerons l'un des (nombreux) cas pour lesquels il s'agit de l'option à privilégier.

Coûts du polymorphisme

Le polymorphisme n'est pas gratuit. L'information requise pour réaliser cette mécanique doit être placée quelque part, et ce pour chaque classe ayant au moins une méthode polymorphique. Dû à cette simple réalité, tout objet polymorphique est plus gros, en mémoire[52], que ne le serait un objet équivalent mais non polymorphique.

Sachant cela, si l'espace supplémentaire requis pour réaliser le polymorphisme à partir de la classe `X` est de `n` *bytes* et si on a besoin de `M` indirections vers des `X` (pensez à une interface personne/ machine contenant des dizaines de milliers de rectangles), la surprime associée au polymorphisme sera d'au moins `M*n` *bytes*. La multiplication, ici, fait mal.

> Quand la dynamique d'un programme requiert du polymorphisme, alors c'est presque toujours la meilleure option (réaliser le même travail sans support du langage coûte presque invariablement plus cher). Quand un programme n'en a pas besoin, toutefois, ce coût est inutile et douloureux.

Classes concrètes

Le concepteur de C++, ***Bjarne Stroustrup***, estime qu'il existe des classes qui ne se prêtent pas au polymorphisme. Sa réflexion est intéressante du fait qu'elle va un peu à l'encontre de bien des dogmes et expose le modèle `OO` comme outil plutôt que comme philosophie.

L'idée est qu'il arrive qu'on souhaite définir une classe non pas à des fins polymorphiques mais bien dans le but de regrouper ensembles comportements et données dans un tout cohérent. Cela peut être pour appliquer un schéma de conception (p. ex. : les singletons [POOv02]), ou pour concevoir une classe dont on ne veut pas vraiment générer des dérivés, qui représente un concept complet en soi.

---

[52] Typiquement, un objet polymorphique contiendra au moins un pointeur de plus, ce pointeur menant vers la table où ses trouvent les adresses des méthodes polymorphiques de l'objet. Le prix à payer peut donc se limiter à quelques *bytes*, mais le problème croît quand on entrepose des milliers ou des millions de ces objets, ce qui est typique des systèmes informatiques normaux.

Les classes pour lesquelles on ne veut vraiment pas de polymorphisme, et pour lesquelles la vitesse est tellement critique que les délais encourus suite à l'emploi d'une méthode virtuelle soient inacceptables – une occurrence rare, mais pas impossible – sont ce qu'on nomme habituellement des ***classes concrètes***.

Maintenant que c'est possible, pourquoi ne pas les qualifier de `final`?

Certains utiliseront aussi classe concrète au sens de classe instanciable, par opposition au concept de classe abstraite que nous examinerons sous peu. C'est d'ailleurs là le sens le plus fréquemment rencontré pour ce vocable.

Cela dit, nous utiliserons ici le sens spécialisé de Stroustrup puisqu'il a pour nous une portée conceptuelle particulièrement intéressante.

Les classes concrètes sont un autre nom pour ce que nous nommons parfois les types valeurs, ceux dont les instances se comportent comme des `int`. Les types valeurs et les types concrets, comme le suggère Stroustrup (pensez au type `std::string` de C++; au type `Monnaie` [POOv00] et, dans une moindre mesure, au type `Integer` tel qu'il apparaît en Java et en C#) ne sont en général pas destinés à être spécialisés, mais bien à être utilisés en tant que tels.

Un exemple de classe concrète possible est donné à droite.

```
class CouleurPrimaire final {
public:
   enum Couleur { rouge, vert, bleu };
private:
   const Couleur couleur;
public:
   CouleurPrimaire(Couleur c) : couleur{c} {
   }
   bool est(Couleur c) const noexcept {
      return couleur==c;
   }
   bool operator==(const CouleurPrimaire &c) const noexcept {
      return est(c.couleur);
   }
   bool operator!=(const CouleurPrimaire &c) const noexcept {
      return !(*this == c);
   }
};
```

Une classe concrète, prise en ce sens, ne prépare pas le terrain pour permettre à ses dérivés d'offrir un comportement polymorphique, et n'expose aucune méthode virtuelle. Elle ne peut donc servir comme racine polymorphique et n'a pas, dans un système, le rôle d'une abstraction pour fins de spécialisations ultérieures.

Ainsi, la classe concrète se veut habituellement compacte et efficace. Un dérivé d'une telle classe obtient surtout des bénéfices semblables à ceux d'un héritage privé ou d'une inclusion par composition.

Un type polymorphique, typiquement, est pensé pour servir d'abstraction spécialisée ultérieurement, et prend un ensemble de dispositions en ce sens.

En particulier, un type concret est normalement fait pour être manipulé directement et offre une sémantique de copie sans effet secondaire, alors qu'un type polymorphique est typiquement manipulé de manière indirecte, donc ses instances son beaucoup moins souvent copiées.

Ceci reflète les usages en C++, qui supporte les types valeurs et les types polymorphiques, et en Java ou dans les langages `.NET`, où la sémantique d'accès indirecte domine : en C++, le constructeur de copie et l'affectation sont des opérations fondamentales, alors qu'en Java ou en C#, la copie est rarement utilisée (et implique de prendre beaucoup de précautions; à cet effet).

Polymorphisme et opérateurs

Les opérateurs se prêtent surtout aux types valeurs, donc aux objets manipulés directement, alors que le polymorphisme se prête surtout aux objets manipulés indirectement.

De manière générale, pour faire en sorte qu'une opération soit polymorphique, il faut qu'elle ait la même interface (la même signature) pour la racine polymorphique *et* pour tous les descendants de cette classe. *Ceci exclut d'office la plupart des opérateurs*, puisque leur signature, implicitement, comporte un opérande qui est différent pour tous les descendants[53].

Se voulant viable pour une hiérarchie de classes toute entière, une méthode polymorphique sera en général une méthode sans paramètre, ou munie de paramètres assez abstraits pour être viables pour l'ensemble des classes auxquelles elle s'appliquera.

Quelques exemples :

- une méthode `deplacer(Destination)` au sens de déplacer l'instance propriétaire vers une destination donnée[54];
- une méthode `agir()`, qui permet à l'instance propriétaire d'agir (selon un contexte qu'elle connaît bien);
- une méthode `calculer_interets()`, qui utilisera une formule viable selon le type précis de l'instance propriétaire (client *Privilège*, client *Or*, client *Argent*, *etc.*);
- un destructeur, qui permet à l'instance à détruire de libérer les ressources qu'elle aura alloué de la manière la plus précise possible; *etc*.

Les opérateurs, eux, sont rarement utilisés de cette manière. Typiquement, un programme exprimant `c = a + b;` aura au préalable déterminé les types de `a`, de `b` et de `c`.

Autres domaines d'application clairs du polymorphisme : tous les cas où l'on désire y aller d'*héritage d'interfaces* (plus bas).

[53] L'opérande de gauche (dans la plupart des cas) est un paramètre implicite des opérateurs, du moins quand ceux-ci sont déclarés sous forme de méthode d'instance.

[54] Question: pourquoi est-on en droit de présumer qu'on n'ait pas besoin d'indiquer de quel endroit l'instance propriétaire entreprend son déplacement?

Polymorphisme et constructeurs

Il existe deux familles de méthodes pour laquelle on n'a même pas le droit d'appliquer le polymorphisme : il s'agit des *constructeurs* et des méthodes de classe (car le polymorphisme se base sur un sujet, `this`, que les méthodes de classe n'ont pas).

Le polymorphisme est la possibilité d'utiliser un pointeur vers *quelque chose d'abstrait*, et d'utiliser la version *la plus concrète possible* d'une méthode donnée, étant donné l'objet effectivement pointé. Tout cela présume bien sûr l'existence au préalable d'un objet. *Un constructeur polymorphique n'a pas de sens*, du moins pas au sens communément entendu du terme polymorphisme[55].

Lorsque le besoin de mettre en place une mécanique polymorphique de construction, on aura souvent recours au schéma de conception **Fabrique** [POOv02].

Polymorphisme et vitesse

Bien que la mécanique du polymorphisme en C++ soit très efficace, elle entraîne un très petit délai de par le besoin qu'a cette mécanique de passer (à l'interne) par une table d'adresses pour trouver la bonne méthode à utiliser.

Il est possible que, pour des applications n'utilisant pas de polymorphisme, ce coût (si petit soit-il) s'avère inacceptable. De telles applications existent, surtout pour des systèmes critiques (comme par exemple les systèmes embarqués ou les systèmes en temps réel). C'est pour assurer à ces applications aussi le meilleur soutien possible que l'utilisation de polymorphisme est une option en C++, pas une obligation.

Chaque invocation polymorphique, peu importe le langage, implique au minimum une indirection à travers une table d'adresses. Cette opération peut sembler lente et complexe, mais les compilateurs OO peuvent réduire cette recherche à l'équivalent d'un accès supplémentaire à un pointeur.

C'est donc une mécanique *presque* aussi efficace que celle reliée à un appel *normal* de fonction ou de méthode; le polymorphisme tel qu'implanté par un compilateur C++ est *beaucoup* plus efficace que toute implémentation manuelle reposant sur une sélective ou toute autre manœuvre semblable.

Même sous Java et C#, le polymorphisme pris en charge à la compilation est un schéma de conception implémenté de manière optimale (dans les circonstances). Un appel polymorphique est plus lent qu'un appel direct, mais sera mieux réalisé par votre compilateur que par vous-même.

---

[55] Cela dit, voir [POOv03] sur la persistance des objets pour une variante sur cette réflexion et sur des pistes de solutions pour un problème qui ressemble à celui, un peu mal posé, de constructeur polymorphique.

Classes terminales

Une classe sera terminale quand ses concepteurs ne souhaitent pas permettre qu'on en dérive. Typiquement, une telle classe n'aura pas été prévue pour qu'on spécialise ses méthodes, et permettre sa surcharge pourrait entraîner des conséquences graves dans un programme. À titre d'exemple, la classe `String` de Java et la classe `string` de C# sont toutes deux terminales, car leur omniprésence dans les programmes entraînerait une gamme de conséquences néfastes si les programmeuses et les programmeurs pouvaient modifier leur comportement. En C++, la classe `std::string` n'est pas terminale au sens du langage, mais en dériver est presque nécessairement une très mauvaise idée puisqu'elle n'offre aucune méthode virtuelle.

En Java et dans les langages `.NET`, le concept de classe terminale est appuyé au niveau du langage. En Java, une classe est terminale si elle est qualifiée **`final`**; C++ utilise le même mot clé mais le positionne ailleurs sur le plan grammatical (voir ci-dessous). En C#, on obtient une classe terminale à l'aide du mot clé **`sealed`** alors qu'en `VB.NET`, le mot clé à utiliser pour obtenir une classe terminale est **`NotInheritable`**.

`VB.NET` offre aussi le mot clé **`NotOverridable`** pour bloquer toute tentative de polymorphisme sur une méthode donnée, ce qui correspond aux méthodes qualifiées `final` ou `sealed` en Java ou en C# (respectivement).

En C++, depuis C++ `11`, le mot clé contextuel **`final`** peut être apposé à une classe terminale. Son usage va comme suit :

```
class X final { // on ne pourra dériver de X
   // ...
};
```

Il est aussi possible d'empêcher la spécialisation ultérieure (par des enfants) d'une méthode polymorphique en la qualifiant `final` :

```
class B {
   // ...
public:
   virtual int f();
   virtual ~B();
};
class D : public B {
   // ...
   int f() final; // les enfants de D ne pourront pas spécialiser B::f
};
```

Il n'est par contre pas possible de qualifier `final` une méthode qui ne serait pas polymorphique au préalable, ce qui est (à mon avis) quelque peu agaçant.

Véritables cas de polymorphisme

Une fois couverts ces cas pour lesquels le polymorphisme n'est pas indiqué, dans quelles circonstances le polymorphisme est-il une bonne option?

La réponse est simple : dans **beaucoup** de cas.

En fait, toute situation où une classe peut servir d'abstraction pour d'autres et où les comportements peuvent être raffinés se prête au polymorphisme.

Au quotidien, nous faisons de telles abstractions sans arrêt : « Fais-moi un joli dessin » (abstraction au sens de `dessiner()` invoqué d'un dessin ou d'une personne), « Il serait sage de tondre la pelouse » (à chacune et à chacun son approche), « J'ai concocté cette recette… » (l'individu a subjectivement concocté un petit plat divin), « Fais ce que tu dois faire » (action subjective à partir d'une abstraction), *etc*.

On peut faire usage de polymorphisme à chaque fois qu'on accepte d'élever le niveau du discours. En discutant en des termes plus abstraits, et en déplaçant la responsabilité de l'implémentation des comportements vers les objets véritablement actifs, nous procédons chaque fois par polymorphisme. Nous verrons encore plusieurs cas d'application du concept dans les pages qui suivent.

Exercice – Série 03

EX00 – Supposons nos classes Carre, Rectangle et Triangle.

Que devrait-on faire pour que le programme suivant affiche un carré par défaut, puis un rectangle de 6 par 9, et finalement un triangle de hauteur 11?

```
int main() {
   const int NB_FORMES = 3;
   Forme *p[NB_FORMES] = {
      new Carre,
      new Rectangle{6, 9},
      new Triangle{11}
   };
   for (int i = 0; i < NB_FORMES; i++) {
      p[i]->dessiner();
      // delete p[i]
   }
}
```

Note :	bien que ceci ne soit qu'un exercice, je vous invite *fortement* à faire ces modifications; elles vous seront utiles à très brève échéance.

EX01 – Présumons la fonction afficher() ci-dessous :

```
#include <iostream>
using std::ostream;
void afficher(ostream &os, const Affichable &aff) {
   aff.afficher(os);
}
```

Rédigez chacune des classes suivantes :

- la classe Affichable, racine polymorphique pour nos fins, dont la méthode virtuelle afficher() prend un ostream par référence et affiche "Surchargez-moi s.v.p";
- la classe Patate, dérivant publiquement de la classe Affichable, et dont la méthode afficher() prend un ostream par référence et affiche "Je suis une patate";
- la classe Carotte, dérivant publiquement de la classe Affichable, et dont la méthode afficher() prend un ostream par référence et affiche "Je suis une carotte".

Ajoutez un programme principal qui instancie une Patate et une Carotte et applique sur chacun de ces objets la fonction globale afficher().

Réflexion

Que pensez-vous de la méthode virtuelle Affichable::afficher(), donc de la version de la méthode afficher() offerte par la classe Affichable?

Le mot clé `virtual`

Le mot clé `virtual` est porteur d'un sens particulier et essentiel dans la dynamique des systèmes OO développés en C++. Cette section explore quelques-unes de ses applications[56], chacune méritant un examen à part entière.

Dans le cas d'un destructeur

Certains prétendent[57] que le mot clé `virtual` devrait *toujours* être appliqué aux destructeurs. **C'est faux**. Cependant, si une classe exposer au moins une méthode virtuelle, alors sont destructeur devrait, lui aussi, être virtuel. Pourquoi donc?

La raison la plus simple est qu'utiliser un destructeur virtuel fera en sorte d'assurer qu'au moment de la destruction d'une instance, le bon destructeur soit toujours sollicité. Ceci est le plus criant lorsqu'on utilise un enfant à travers un pointeur vers l'un de ses ancêtres (à droite).

Dans l'exemple à droite, `b` pointe sur un `Derive`, et `new Derive` invoque le constructeur par défaut de `Derive`, mais `delete b;` invoque le destructeur de Base du fait que `Base::~Base()` n'est pas virtuel. Conséquemment, si `Derive` compte sur son destructeur pour nettoyer les ressources placées sous sa gouverne, ce nettoyage n'aura jamais lieu!

```
struct Base {
   virtual int f(); // peu importe
   // ~Base() devrait être virtuel
   ~Base();
};
struct Derive : Base {
   ~Derive();
};
int main() {
   // création d'un Derive
   Base* b = new Derive;
   // appelle ~Base() mais pas ~Derive()!
   delete b;
}
```

La recette : si une classe possède au moins une méthode virtuelle, elle devient susceptible de servir d'abstraction pour ses dérivés, et devrait alors exposer un destructeur virtuel.

Si le destructeur d'une classe exposant un comportement polymorphique n'est pas virtuel, alors il devient possible que des ressources allouées à la construction ne soient pas libérées à la destruction. Si un objet avait alloué des ressources manuellement, alors ces ressources ne seront pas libérées. Ceci peut avoir de très, *très* fâcheuses conséquences.

Dans [EffCpp], item 7, ***Scott Meyers*** nous recommande de ne pas dériver d'une classe ne possédant pas de destructeur virtuel. Sans aller jusque-là, il se trouve qu'en pratique, une classe n'exposant pas de méthode virtuelle est rarement une bonne abstraction pour ses enfants. On n'utilise pas les abstractions polymorphiques et les types valeurs de la même manière.

---

[56] L'une des applications du mot clé `virtual` est intimement liée au concept d'héritage multiple. Conséquemment, nous l'aborderons en temps et lieu.

[57] Pour un bémol, voir la section ***Classes concrètes***, plus haut.

En C++, le mot `virtual` devrait apparaître seulement dans la déclaration (*pas dans la définition*) d'une méthode pouvant être utilisée par polymorphisme, souvent dans la classe racine du concept représenté par le polymorphisme. Par exemple, si toute forme peut être dessinée, et il faut donc que la méthode `dessiner()` soit `virtual` *au moins pour* `Forme`.

Bien que ce ne soit pas requis, on peut spécifier sans problème une méthode comme étant `virtual` dans une classe dérivée même si elle l'était déjà dans la classe parent. Au pire, la redondance sera éliminée à la compilation.

Le polymorphisme sur une méthode donnée est applicable à partir de la première classe qui, dans une hiérarchie donnée, spécifie cette méthode comme étant virtuelle. À partir de cette classe, la méthode en question devient polymorphique pour l'ensemble des descendants.

Exprimé simplement : le fait qu'un comportement soit polymorphique est une caractéristique héritée d'un parent par ses enfants, de manière transitive (mais dans le respect des clauses d'héritage public, protégé et privé).

```
struct B {
   virtual int f() const {
      return 3;
   }
   virtual ~B() = default;
};
struct D : B {
   virtual int f() const {
      return 4;
   }
};
class DD : private D {
   virtual int f() const {
      return 5;
   }
public:
   B* grand_parent() {
      return this;
   }
};
#include <iostream>
int main() {
   using std::cout;
   DD *pDD = new DD;
   B *pB = pDD-> grand_parent();
   cout << pB->f();
   delete pB;
}
```

L'exemple à droite est quelque peu pervers mais illustre bien la réalité de la transitivité du polymorphisme dans un contexte d'héritage qui transcende les cas simples et évidents.

- Tout `DD` est (indirectement, en passant par la classe `D`) un `B`, mais de manière telle que seul un `DD` le sait (par héritage privé).
- La classe `B` y expose un comportement polymorphique (méthode `f()`), que `DD` raffine (avec une méthode privée par-dessus le marché).
- La classe `DD` offre une méthode publique permettant d'obtenir indirectement le `B` caché dans un `DD`, ce par quoi le programme principal parvient à invoquer `f()` (publique sur un `B` même si elle est privée sur un `DD`) et obtient le `f()` de `DD` par polymorphisme.

Non, je ne recommande pas de programmer ainsi, mais l'illustration montre la souplesse du modèle.

Mot clé contextuel `override`

Depuis C++ 11, il est possible, sans être obligatoire, pour une classe dérivée d'expliciter son intention de spécialiser une méthode virtuelle en lui apposant le mot-clé contextuel `override` :

```
class B {
   // ...
public:
   virtual int f();
   virtual ~B();
};
class D : public B {
   // ...
   int f() override; // D souhaite spécialiser le f() d'un ancêtre
};
```

Avec la qualification `override`, un compilateur signalera une erreur si l'enfant sembler chercher à spécialiser un service qui n'existe pas chez ses ancêtres, ou qui n'est pas polymorphique. Ceci permet de détecter des erreurs, résultant souvent de distractions (p. ex. : erreur dans le nom d'une méthode, oubli d'une qualification `const`, choix du mauvais service), et peut simplifier l'entretien du code.

Dans d'autres langages

Le polymorphisme est l'un des aspects les plus fondamentaux de la POO, mais la manière par laquelle il s'exprime et s'exploite tend à varier d'un langage à l'autre.

En Java, le mot clé `virtual` n'apparaît pas car toutes les méthodes d'instance, outre les constructeurs et les méthodes qualifiées `final`, sont polymorphiques. Ceci complique d'ailleurs la tâche de l'optimisateur accompagnant le compilateur Java, du fait que certaines stratégies d'optimisation ne sont possibles que pour les méthodes dont l'invocation éventuelle est connue à la compilation (ce qui est en quelque sorte l'inverse du polymorphisme).

Java compense pour cette déficience par l'intégration d'un profileur sophistiqué à la JVM, qui réalise certaines optimisations sur la base du comportement effectif des programmes.

En C#, comme en C++, les méthodes ne seront polymorphiques que si on leur applique la qualification `virtual`. De plus, une classe dérivée voulant bénéficier du polymorphisme doit l'indiquer en appliquant le mot **`override`** sur la déclaration de la méthode dans la classe dérivée.

> La présomption faite ici est qu'une équipe de développement risque moins ainsi d'appliquer un polymorphisme non désiré de manière accidentelle si l'utilisation d'un mot clé pour ce faire est obligatoire. De même, ceci permet de dépister certaines erreurs de design, comme une tentative de surcharge d'une méthode qui, à la base, n'aurait pas été déclarée virtuelle.
>
> Forcer le code utilisateur client à clarifier ses intentions est une signature des langages .NET, reflet d'une philosophie du développement et de l'évolution du code que certains approuveront, d'autres moins[58].

```
namespace DémoPolymorphisme
{
   class B
   {
      public virtual int valeur()
      {
         return 3;
      }
   }
   class DPoly : B
   {
      public override int valeur()
      {
         return 4;
      }
   }
   class DNonPoly : B
   {
      public int valeur()
      {
         return 5;
      }
   }
}
```

Dans l'exemple à droite, la méthode `valeur()` de `DPoly` aura un comportement polymorphique à travers une référence à `B` car elle spécifie `override` dans sa déclaration. En retour, la méthode `valeur()` de `DNonPoly` n'aura pas un comportement polymorphique.

[58] On vise traditionnellement à responsabiliser les concepteurs de modules destinés à être réutilisés pour les motiver à publier des interfaces stables et réutilisables. C'est une vision qui se veut ***optimiste***. Conséquemment, la tradition suggère qu'une classe parent ou qu'un prototype de méthode soit le lieu déterminant les règles éventuelles d'utilisation et que ce lieu soit aussi stable que possible. Les langages .NET, eux, demandent au code client de préciser *à l'utilisation* ses attentes, ce qui permet de déterminer les incohérences entre le contrat exposé et la vision qu'en a le code client. C'est une vision ***pessimiste*** du développement, qui présume que des modifications se glisseront dans les interfaces publiques suite à leur publication. Les *aficionados* de la vision pessimiste mettront de l'avant que leur position est pragmatique et tient compte de la faillibilité des programmeurs, et rappelleront que plusieurs designs OO ont échoué suite à une conception déficiente. Ceux privilégiant la vision optimiste souligneront que la vision pessimiste favorise la conception déficiente.

```
B p0 = new B(),
  p1 = new DPoly(),
  p2 = new DNonPoly();
System.Console.Write(
   "{0} {1} {2}",
   p0.valeur(), p1.valeur(), p2.valeur()
);
```

Ainsi, un extrait de programme comme celui à droite, faisant référence aux classes ci-dessus, affichera `3 4 3`, pas `3 4 5`.

```
Namespace DémoPolymorphisme
   Public Class B
      Public Overridable Function GetVal() As Integer
         GetVal = 3
      End Function
   End Class
   Public Class DPoly
      Inherits B
      Public Overrides Function GetVal() As Integer
         GetVal = 4
      End Function
   End Class
   Public Class DNonPoly
      Inherits B
      Public Overloads Function GetVal() As Integer
         GetVal = 5
      End Function
   End Class
   Public Class Test
      Public Shared Sub main()
         Dim p0 As B = New B
         Dim p1 As B = New DPoly
         Dim p2 As B = New DNonPoly
         System.Console.Write("{0} {1} {2}", _
            p0.GetVal(), p1.GetVal(), p2.GetVal())
      End Sub
   End Class
End Namespace
```

En **VB.NET**, la stratégie est encore plus explicite qu'avec C# et trois mots clés interviennent :

- une méthode pouvant servir à des fins polymorphiques doit être initialement déclarée **Overridable**;
- une surcharge à fins polymorphiques doit être spécifiée **Overrides**; et
- une surcharge à des fins non polymorphiques, cachant carrément la méthode initiale, doit être spécifiée **Overloads**.

Détail particulier à VB.NET: le concept de *Shadowing* (que je n'ose pas traduire par *ombrage*), soit l'acte de cacher un nom (un attribut, une méthode) par un autre. En C++, Java et C#, plusieurs noms identiques peuvent exister dans la mesure où deux noms identiques ne sont pas déclarés dans le même bloc. En cas d'usage reposant seulement sur le nom en question, alors le sens le plus local prime sur les sens moins locaux.

En VB.NET, déclarer deux noms identiques de manière à ce qu'un nom en cache un autre demande qu'on ait recours au mot clé **Shadows** pour le nom qui dominera.

L'exemple à droite en illustre un usage avec deux classes en situation d'héritage et un nom (N) assujetti à deux usages (et à deux types!) distincts.

Le recours à Shadows rend Dérivé.N légal.

```
Namespace Test
   Public Class Base
      Public N As Integer = 100 ' taille d'un tableau?
      ' etc.
   End Class
   Public Class Dérivé
      Inherits Base
      Public Shadows N As String = "Fred" ' N au sens de «Nom»?
      ' etc.
   End Class
   Public Class Test
      Public Shared Sub main ()
         Dim p0 As Base = New Dérivé ' p0 traité comme un Base
         Dim p1 As Dérivé = New Dérivé ' p1 pris comme un Dérivé
         System.Console.WriteLine ("{0} {1}" p0.N, p1.N)
      End Sub
   End Class
End Namespace
```

Abstraction

Pourquoi cette section?

L’héritage mène à penser les classes sous forme hiérarchique, du plus général au plus spécifique. Le polymorphisme, lui, mène à rédiger des programmes en termes des abstractions les plus élevées à généraliser le code client et à spécialiser le travail des entités actives que sont les objets. L’étape naturelle découlant de cette démarche est d’approcher les problèmes complexes à partir d’abstractions pures et de contrats.

Élevons maintenant le niveau d’abstraction de notre discours et profitons à plein des capacités de généralisation et d’abstraction du modèle OO, en posant cette question en apparence toute simple : *que signifierait l’opération* `dessiner()` *pour une instance de* `Forme`*?*

Ne nous méprenons pas : la question est claire pour un dérivé de `Forme` (comme `Triangle` par exemple), mais a-t-elle du sens pour une `Forme` en tant que telle? De quoi aura aurait l’air une `Forme` *abstraite* si on la dessinait?

⇒ Si on ne peut pas raisonnablement ***exprimer*** une idée, comment pourrait-on la ***définir***?

Méthodes abstraites

Il arrive fréquemment qu’on rencontre une méthode telle que :

- toutes les classes dérivées d’un ancêtre commun doivent en avoir une définition (ici : on veut que toute `Forme` puisse être dessinée);
- on veut traiter tous les dérivés de manière générale comme des spécialisations de cet ancêtre commun (ici : on veut exprimer le code client comme opérant sur diverses instances de `Forme`, sans obliger ce programme à descendre au niveau des détails); mais
- pour l’ancêtre en question, pour le cas général, il est impossible de définir ladite méthode, *celle-ci étant trop abstraite* (ici : que signifie dessiner une `Forme` en un sens général?).

C’est précisément le cas rencontré ici. On dira que la méthode `dessiner()` de `Forme` est ***abstraite***. Cela signifie que, dans la classe `Forme`, nous allons donner le prototype (la déclaration) de la méthode `dessiner()`, puisque nous voudrons l’utiliser pour fins de polymorphisme, mais nous allons aussi faire en sorte de ne pas définir cette méthode pour la classe `Forme` elle-même.

La syntaxe lors de la déclaration d’une méthode abstraite sera, si on prend l’exemple de la méthode `dessiner()`, celle visible à droite.

```
class Forme {
   // ...
   virtual void dessiner() const = 0;
   virtual ~Forme();
   // ...
};
```

En C++, on dira aussi d’une telle méthode qu’elle est **pure virtuelle**. Cependant, le terme consacré et reconnu dans plusieurs langages est **méthode abstraite**, donc c’est cette forme que nous privilégierons.

La raison d'être d'une méthode abstraite s'explique comme suit :

> Une méthode abstraite est toujours polymorphique (une méthode abstraite **doit** être spécialisée par les enfants). Une méthode polymorphique n'est pas nécessairement abstraite (une méthode polymorphique qui n'est pas abstraite **peut** être spécialisée par les enfants).

- on veut que la classe où une méthode abstraite est déclarée permette l'emploi indirect, polymorphique, de cette méthode. Par exemple, on veut que `Forme` expose la méthode `dessiner()` car on veut pouvoir `dessiner()` toute `Forme` en tant que `Forme`;
- on ne veut par contre pas qu'il soit possible d'utiliser la méthode directement à travers une instance de la classe où elle est déclarée. On ne veut pas qu'il soit possible de compiler le code à droite *car cet extrait de programme n'a pas de sens*.

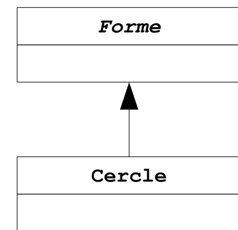
```
Forme f;
// ???
f.dessiner();
```

Une méthode abstraite est un ***contrat*** pour tous les descendants de la classe où elle est déclarée. C'est une méthode que chacun de ces descendants devra implanter s'il veut échapper à une contrainte drastique : celle de devenir une ***classe abstraite***.

Classes abstraites

⇒ Toute classe ayant au moins une méthode abstraite est considérée abstraite.

Ainsi, si `Forme` a la méthode abstraite (pure virtuelle) `dessiner()`, alors `Forme` est une classe abstraite. La notation `UML` utilisée pour dénoter une classe abstraite demande d'indiquer son nom en caractères italiques dans les diagrammes.



Dans le schéma à droite, `Forme` est abstraite mais `Cercle` ne l'est pas. On pourrait indiquer le nom de la méthode abstraite (`dessiner()`) en italiques dans `Forme` et sans italiques dans `Cercle` pour clarifier encode plus le propos.

Une classe abstraite ne peut être instanciée. Ceci fait en sorte qu'il sera impossible de déclarer une `Forme` de l'une ou l'autre des manières suivantes.

```
// Forme expose au moins une méthode
// abstraite, donc Forme est abstraite
class Forme {
   // ...
public:
   virtual void dessiner() const = 0;
   virtual ~Forme();
   // ...
};
int main() {
   // illégal!
   Forme f;
   // illégal!
   Forme *pf = new Forme;
}
```

Un dérivé d'une classe abstraite est considéré abstrait s'il ne définit pas toutes les méthodes abstraites de son parent. *Si on veut créer la classe `Triangle` qui dérive de `Forme` et si on veut éventuellement instancier un `Triangle`, il faudra donc définir la méthode `dessiner()` dans la classe `Triangle`.*

> Il peut sembler étrange, à première vue, d'écrire des classes qu'on ne peut instancier, ou d'imposer à un programme des contraintes. Songez toutefois au code client : s'il obtient une indirection vers une `Forme`, il a la *certitude* que cette indirection implémente `dessiner()`, sinon elle n'aurait pas pu être instanciée. Les classes abstraites imposent un fardeau aux concepteurs des classes, mais offrent des garanties au code client.

On peut manipuler une indirection dont le type est une classe abstraite (une `Forme &` ou un `Forme *`, par exemple), utilisant les méthodes déclarées pour cette classe (abstraites ou non), dans la mesure où les instances effectivement pointées sont des instances de dérivés qui ne sont pas, eux-mêmes, abstraits (donc dans la mesure où, au bout de la `Forme &` ou du `Forme *`, se trouve une instance d'une classe dérivée de `Forme`, disons `Triangle`, qui n'aurait plus la moindre méthode abstraite).

On ne pourra légalement, à travers ce pointeur, utiliser que les méthodes dévoilées par la classe parent; c'est la magie du polymorphisme à l'œuvre!

Le moment de construction d'un objet n'est jamais polymorphique; on connaît toujours le type effectif d'un objet lors de sa construction. À droite, `t` est une instance de `Triangle` créée de manière automatique, sur la pile, alors que `f` n'y mène que de manière indirecte.

De son côté, `pf` pointe au moins sur une `Forme` mais c'est l'opérateur `new` qui enclenche la construction de l'instance, or cet opérateur instancie bel et bien un `Triangle`, pas une `Forme`.

```
// Légal (si Triangle définit la
// méthode dessiner(), bien sûr!)
Triangle t;
Forme &f = t; // Ok
Forme *pf = new Triangle; // Ok
// Dessiner un Triangle, par
// polymorphisme
f.dessiner();
pf->dessiner();
// Essentiel que le destructeur
// de Forme soit virtuel!
delete pf;
```

Interfaces

Une idée clé de la `POO` contemporaine, sur laquelle nous reviendrons lorsque nous discuterons d'héritage multiple (plus bas), est celle des **interfaces**. Bien que ce terme ait une acception générale au sens de visage public d'une classe, il a aussi un sens technique dans le monde `OO`.

Pour situer grossièrement cette idée, nous en donnerons pour le moment une définition simple et opérationnelle, sur laquelle nous reviendrons en temps opportun : *une interface est une classe strictement abstraite, un pur contrat*. Une interface n'a aucun attribut, et toutes ses méthodes sont des méthodes d'instance abstraites.

On utilise les interfaces pour décrire des idées purement opératoires : un `Dessinable` doit savoir se dessiner, un `Stoppable` doit savoir s'arrêter, un `Mobile` doit savoir se déplacer et doit révéler, sur demande, sa position et son orientation, *etc.*

Ce sujet mérite un traitement approfondi, que nous lui accorderons un peu plus loin.

Rôle de l'abstraction

Le rôle d'une abstraction celui d'un modèle permettant d'exprimer des idées générales en se concentrant sur l'essentiel. Les classes abstraites et les méthodes abstraites, en ce sens, proposent un modèle auquel tous leurs dérivés devront se conformer pour qu'ils puissent être instanciés.

Une démarche typique de conception par abstraction sera :

- rédiger la classe `Forme`, sans savoir quelles seront les classes dérivées de `Forme` (et il pourra y en avoir des milliers!) et, en y faisant de `dessiner()` une méthode abstraite, y exprimer que dans notre optique, toute `Forme` doit savoir se dessiner;
- plus tard, déclarer des descendants (directs ou non) publics de `Forme` et implanter la méthode `dessiner()`;
- instancier diverses instances dérivées de `Forme` selon les besoins; et
- invoquer `dessiner()` à travers ces indirections vers des dérivés de `Forme` en ayant pleinement confiance, par définition, que cette méthode sera implémentée, sans savoir ce que cette implémentation sera en pratique.

Parenthèse technique

Comment un compilateur peut-il ainsi deviner le futur, et trouver de lui-même et à l'avance des méthodes qui n'existent pas encore? Comment, par exemple, comprend-il alors qu'il compile `Forme` qu'un jour, il y aura `Cercle` et qu'il devra repérer sa méthode `dessiner()`?

La réponse est simple : *le compilateur ne le sait pas*. Ce que le compilateur sait, par contre, c'est quelle est la signature des méthodes virtuelles de `Forme` (nom, nombre de paramètres et type des paramètres), et dans quel ordre elles sont déclarées dans la classe `Forme`.

Sachant ceci, le compilateur est en mesure de générer, à la compilation, une table de méthodes virtuelles nommée, en C++, la `vtbl`[59] (mais d'autres langages utilisent cette terminologie aujourd'hui consacrée). La disposition des méthodes virtuelles de `Forme` dans cette table respecte l'ordre de leur déclaration pour la classe `Forme`; on ne s'occupe habituellement pas de ces détails, sauf dans les cas où deux technologies distinctes doivent interfacer.

Si `Cercle` dérive de `Forme`, il doit connaître la déclaration de `Forme`. Ceci indique au compilateur la signature de chaque méthode virtuelle y apparaissant et de leur ordre d'apparition. Il ne lui reste plus qu'à remplacer, dans la `vtbl` de chaque `Cercle`, les adresses des méthodes de `Forme` par celles de `Cercle`[60]. Invoquer une méthode de `Forme` par une indirect vers une `Forme` passe par l'adresse indiquée dans sa `vtbl` invoque la méthode qui s'y trouve, soit celle appropriée pour un `Cercle`.

---

[59] Les plus anciens se souviendront qu'on a longtemps utilisé le vocable `VMT`, pour *Virtual Method Table*. En C++, toutefois, cette table sera à bien d'autres choses qu'aux méthodes polymorphiques, alors `vtbl` y est plus approprié que ne l'est `VMT`.

[60] Le `= 0;` utilisé en suffixe de la déclaration d'une méthode abstraite peut être vu comme insérant un pointeur nul dans la table de méthodes virtuelles, permettant ainsi de détecter, dans une classe donnée, l'absence d'une implantation valide pour cette méthode.

Polymorphisme en situation d'instabilité

Il est important, dans les moments où l'identité d'un objet est en fluctuation, donc dans les constructeurs et dans le destructeur, d'éviter d'avoir recours à des méthodes polymorphiques.

Cette contrainte, qui s'avère valide pour tout comportement polymorphique mais est particulièrement frappante dans le cas des méthodes abstraites, peut sembler étrange à première vue. Il est toutefois très important de ne pas y contrevenir.

À titre d'exemple, imaginons cette situation :

- nous concevons la classe `EntierBorne`, qui représente un entier dont la valeur doit se situer entre deux bornes;
- imaginons qu'on veuille laisser chaque classe dérivée spécifier ses propres bornes minimum et maximum (inclusives), de même que ce qu'elle estime être une bonne valeur par défaut;
- imaginons, pour ce faire, qu'`EntierBorne` expose trois méthodes abstraites, nommées `borne_min()`, `borne_max()` et `valeur_defaut()`, forçant ainsi tout dérivé de cette classe à les implanter;
- la méthode `valider()` permettra d'assurer la validité des valeurs passées au constructeur paramétrique[61].

```
class EntierBorne {
public:
   class HorsBornes {};
private:
   int valeur_;
   int valider(int val) {
      if (val < borne_min() || borne_max() < val)
         throw HorsBornes{};
      return val;
   }
public:
   EntierBorne() : valeur_{ valeur_defaut() }{
   };
   EntierBorne(int val) : valeur_{valider(val)} {
   }
   int valeur() const noexcept {
      return valeur_;
   }
protected:
   virtual int borne_min() const=0;
   virtual int borne_max() const=0;
   virtual int valeur_defaut() const=0;
   // ...
};
```

Le choix de déclarer les méthodes abstraites `borne_min()`, `borne_max()` et `valeur_defaut()` est un choix esthétique et politique. Agissant ainsi, les enfants d'`EntierBorne` pourront (`protected`) et devront (abstraites) les surcharger, mais ces méthodes joueront un rôle structurel et ne seront pas visibles pour le code client. D'autres auraient pu choisir de les exposer publiquement.

Nous avons donc, en apparence du moins, mis en place une mécanique permettant de rédiger plusieurs classes représentant des valeurs entières dont les bornes de validité sont connues et de les manipuler comme telles. Il y a pourtant un vice dans notre approche.

[61] Souvenons-nous que seules les méthodes d'instance peuvent être polymorphiques, ce qui explique que `valider_valeur()`, qui aura entre autres besoin d'invoquer `borne_min()` et `borne_max()`, soit aussi une méthode d'instance.

Pour illustrer le problème, imaginons la classe dérivée `Note` qui implante des bornes de `0` et `100`, de même qu'une valeur par défaut de `0`.

Remarquez que les méthodes `borne_min()`, `borne_max()` et `valeur_defaut()` sont déclarées privées ici, ce qui signifie qu'elles ne sont pas accessibles à partir du code client ou des enfants de `Note`. Puisque ce sont des spécialisations des méthodes du parent, `EntierBorne`, ce dernier pourra quand même les invoquer de manière polymorphique (le lieu d'appel est protégé dans le parent, donc accessible au parent, même si l'implémentation, elle, est privée).

Maintenant, examinons l'invocation du constructeur par défaut de `Note`, qui invoquera au préalable le constructeur par défaut d'`EntierBorne`.

```
#include "EntierBorne.h"
class Note : public EntierBorne {
   // ...
   int borne_min() const {
      return 0;
   }
   int borne_max() const {
      return 100;
   }
   int valeur_defaut() const {
      return 0;
   }
public:
   Note() = default;
   Note(int val) : EntierBorne{val} {
   }
   // ...
};
```

Le constructeur du parent `EntierBorne` s'exécutera avant même que n'ait commencé l'existence de cet enfant qu'est l'instance de `Note` en cours de construction. Ce détail est important, et doit être saisi pour bien comprendre les ramifications de ce qui suit.

Le constructeur d'`EntierBorne` utilisera, comme valeur initiale pour `valeur_`, la valeur retournée par la méthode abstraite `valeur_defaut()`.

```
EntierBorne::EntierBorne()
   : valeur_{valeur_defaut()}
{
}
```

L'objectif manifeste est de permettre à l'enfant de donner au parent les consignes propres à sa propre initialisation, or voici le problème : **au moment de la construction du parent, l'enfant n'existe pas encore**, pas plus que ses méthodes virtuelles (abstraites ou non). `EntierBorne`, dans son constructeur par défaut, invoque une méthode d'un enfant qui n'existe même pas!

Pour que le polymorphisme s'applique correctement, l'objet doit exister en totalité. ***Il faut donc, en C++, éviter l'emploi de méthodes virtuelles dans un constructeur***, quand l'objet est en cours de construction (moment où il est probable qu'il soit incomplètement construit) ***ou dans un destructeur*** (moment où il est probablement que des parties de l'objet n'existent plus).

De manière plus générale, imaginons le programme ci-dessous. Une classe parent, nommée `X`, y expose une méthode publique virtuelle nommée `f()`, et appelle cette méthode dans son constructeur. Une classe `Y`, dérivée de `X`, raffine la méthode `f()`. Enfin, un programme instancie `Y`, ce qui appellera le constructeur du parent `X` et invoquera la méthode `f()`.

Une question se pose, beaucoup moins anodine qu'il n'y paraît : quelle sera la version de la méthode `f()` qui sera invoquée lors de l'instanciation de `Y`?

En C++, la réponse peut surprendre : en effet, c'est la version de `X` qui sera appelée, et ce même si `f()` est virtuelle et si on instancie un `Y`.

La raison pour cet état de fait est que l'appel est fait dans un constructeur. En instanciant une classe enfant, on instancie d'abord ses parents (puis leurs parents, et ainsi de suite de manière récursive). L'instanciation des parents précède le début de l'instanciation de l'enfant, ce qui transparaît dans la syntaxe du langage (les appels aux constructeurs des parents sont logés avant l'accolade ouvrante du constructeur de l'enfant).

Au moment où le constructeur de `X` est invoqué, il n'y a pas encore de `Y` qui existe. Clairement, *à ce moment de son existence*, l'objet est *véritablement* un `X`!

```
#include <iostream>
using std::cout;
struct X {
   virtual void f() const {
      cout << "X::f()";
   }
   X() {
      f();
   }
};
struct Y : X {
   void f() const override {
      cout << "Y::f()";
   }
};
int main() {
   Y y; // ???
}
```

Une situation analogue apparaît d'ailleurs dans les destructeurs, alors que l'identité d'un objet passe de la classe enfant à ses classes parentes de manière récursive.

Le polymorphisme, pour fonctionner, repose conceptuellement sur une inférence dynamique du type effectif d'un objet auquel on réfère indirectement, par un pointeur ou par une référence. Pour que cette mécanique soit sans effets secondaires, il est essentiel de pouvoir compter sur un objet dont l'identité est stable.

Méthodes abstraites implémentées

Beaucoup de gens l'ignorent, mais il est possible en C++ d'implémenter une méthode même si elle est à la base déclarée abstraite.

Par exemple, dans la classe `Message` proposée à droite, la méthode `afficher()` est abstraite, ce qui impose à tous les dérivés de `Message` l'obligation de l'implémenter.

L'idée peut être, par exemple, que toutes les instances de `Message` d'un programme seront regroupées dans un même tableau et seront affichées sur un flux selon un format variant en fonction des catégories de message. Un dérivé de `Message` serait alors écrit pour chaque famille de messages et spécifierait le format d'affichage à utiliser.

```
#ifndef MESSAGE_H
#define MESSAGE_H
#include <iosfwd>
#include <string>
class Message {
   // ...
public:
   std::string nom() const;
   virtual void afficher(std::ostream&) = 0;
   virtual ~Message();
   // ...
};
#endif
```

Les concepteurs de `Message` peuvent avoir pensé leur classe de manière à imposer à toutes les programmeuses et à tous les programmeurs qui en créeront des dérivés une réflexion sur le format du message à afficher.

Cependant, rien ne les empêche d'avoir prévu une sorte de message par défaut, entre autres pour ces quelques cas où la réflexion sur le format requis mènerait au constat qu'un tel message serait suffisant.

Une implémentation simple est proposée à droite, à titre d'exemple. Notez que la méthode demeure abstraite, et que les enfants ont malgré tout l'obligation de la spécialiser.

```
#include "Message.h"
#include <iostream>
using namespace std;
// ...
void Message::afficher(ostream &os) {
   os << nom() << endl;
}
```

Prenons maintenant, à titre d'exemple, une petite classe nommée `MessageBanal`, qui représente un message très, très simple. Devant s'intégrer à notre modèle, cette classe dérivera publiquement de `Message` et, respectant le contrat dont elle hérite ainsi, implémentera évidemment sa propre version de la méthode `afficher()`.

Cependant, dû à la banalité du travail à accomplir, elle s'en remettra à la version par défaut implémentée par `Message`.

```
#ifndef MESSAGE_BANAL_H
#define MESSAGE_BANAL_H
#include "Message.h"
#include <iosfwd>
class MessageBanal : public Message {
   // ...
   void afficher(std::ostream&) override;
   // ...
};
#endif
```

```
#include "MessageBanal.h"
#include "Message.h"
#include <iostream>
using std::ostream;
// ...
void MessageBanal::afficher(ostream &os) {
   Message::afficher(os);
}
```

Pour y arriver, `MessageBanal` invoquera la méthode `afficher()` de son parent. Ceci montre de quelle manière une implémentation offerte pour une méthode abstraite peut être utile aux enfants, même quand ceux-ci doivent la spécialiser.

Destructeurs abstraits

```
// Base est abstraite car
// Base::~Base() est abstraite
struct Base {
   // ... est-ce légal?
   virtual ~Base() = 0;
};
```

On pourrait être tenté de proposer, dans une classe destinée à servir de racine polymorphique, d'insérer un destructeur abstrait, ce qui pourrait ressembler à l'extrait de programme proposé à droite.

On pourrait d'ailleurs raisonner ce geste en indiquant :

- que l'on souhaite, pour nos propres raisons (quelles qu'elles soient), que la classe `Base` soit vouée à un usage polymorphique;
- qu'on veut que les dérivés effectivement créés (donc qui ne sont pas abstraits) aient *nécessairement* un destructeur.

Cela semble logique, du moins à première vue. Cela dit, il est illégal de déclarer un destructeur abstrait (à moins de fournir en même temps une implémentation par défaut). Cela prend tout son sens quand on réfléchit à la mécanique de destruction d'un objet.

En effet, si `D` dérive de `Base`, alors :

- détruire une instance de `D` impliquera évidemment détruire son parent `Base`; donc
- le destructeur du parent sera éventuellement appelé *pour lui-même*, même s'il est virtuel;
- il est donc clair qu'une définition doit exister pour un destructeur polymorphique, puisqu'elle sera invoquée.

Exposer un destructeur virtuel dans toute classe possédant au moins une méthode virtuelle est presque nécessairement la meilleure chose à faire. Si on ne peut penser au code à insérer dans le destructeur d'une racine polymorphique, on peut en faire une opération abstraite, mais seulement dans la mesure où une implémentation par défaut (au pire, un bloc vide) est offerte.

Dans d'autres langages

En Java, une méthode abstraite doit être qualifiée du mot clé `abstract`. Une classe possédant au moins une méthode abstraite doit être explicitement qualifiée `abstract` (en C++, exposer une méthode abstraite signifie *implicitement* être une classe abstraite).

Comme en C++, l'abstraction est une caractéristique héritée. Une classe dérivant d'un parent abstrait demeure abstraite jusqu'à ce qu'elle implémente les méthodes qui étaient abstraites pour son parent.

Dans l'exemple à droite, la classe `Forme` est spécifiée `abstract` parce que sa méthode `dessiner()` est elle-même spécifiée `abstract`. De son côté, la classe `Carré` n'est pas spécifiée `abstract` parce qu'elle définit la méthode `dessiner()`.

Souvenons-nous que toutes les méthodes d'instance en Java (hormis les constructeurs) sont polymorphiques.

```
abstract class Forme {
   // ...
   public abstract void dessiner();
}
class Carré extends Forme {
   private static final int HAUTEUR = 5;
   private static final int LARGEUR = HAUTEUR;
   // ...
   public void dessiner() {
      for (int i = 0; i < HAUTEUR; i++) {
         for (int j = 0; j < LARGEUR; j++) {
            System.out.print("*");
         }
         System.out.println();
      }
   }
}
public class Z {
   public static void main(String[] args) {
      Carré c = new Carré();
      c.dessiner();
   }
}
```

Java est une créature différente de C++ sur le plan du polymorphisme. En effet, hormis le cas particulier des constructeurs, les méthodes d'instance de Java sont toutes virtuelles, sans exception. Sachant cela, si la consigne importante en C++ d'éviter les appels de méthodes virtuelles dans les constructeurs tenait aussi pour Java, alors on ne pourrait y invoquer aucune méthode d'instance dans un constructeur, ce qui serait à tout le moins désagréable.

Java se limite à l'héritage simple, et l'invocation du constructeur du parent se fait par un appel à une méthode spéciale nommée `super()` pouvant recevoir des paramètres (pour utiliser d'autres constructeurs du parent que le constructeur par défaut).

Java oblige toute classe dérivée à faire de l'invocation de `super()` la première opération de ses propres constructeurs. Dans le cas contraire, un appel à `super()` sans paramètre est inséré automatiquement au début du constructeur de l'enfant, et les appels subséquents à `super()` dans ce constructeur entraînent des erreurs de compilation.

La différence syntaxique entre Java et C++ relève d'une différence conceptuelle : en C++, on construit les parents avant de construire l'enfant, alors qu'en Java on fait de la construction du parent la première étape dans la construction de l'enfant.

L'enfant existe déjà, mais ses attributs ne sont pas encore initialisés; la prudence reste de rigueur.

```
class X {
   void f() {
      System.out.println("X.f()");
   }
   X() {
     f ();
   }
}
public class Y extends X
{
   void f() {
      System.out.println("Y.f()");
   }
   Y() {
      super();
   }
   public static void main(String [] args) {
      Y y = new Y();
   }
}
```

Java n'offre pas de destructeurs, donc le problème de la fluctuation identitaire en période de destruction d'un objet ne se pose pas dans ce langage.

En C#, la situation est presque identique à celle rencontrée en Java.

Le mot clé pour qualifier une méthode ou une classe d'abstraite demeure `abstract` et les circonstances et règles son précisément les mêmes que pour Java.

```
namespace z
{
   abstract class Forme
   {
      // bla
      public abstract void Dessiner();
   }
   class Carré : Forme
   {
      private const int HAUTEUR = 5;
      private const int LARGEUR = HAUTEUR ;
      // bla
      public override void Dessiner()
      {
         for (int i = 0; i < HAUTEUR; i++)
         {
            for (int j = 0; j < LARGEUR; j++)
               System.Console.Write("*");
            System.Console.WriteLine();
         }
      }
   }
   public class Z
   {
      public static void Main(string[] args)
      {
         Carré c = new Carré();
         c.dessiner();
      }
   }
}
```

Le langage C# est, en ce qui a trait au polymorphisme, un hybride entre Java et C++; un hybride qui a des qualités, évidemment, mais fait un peu de magie pour les mettre en valeur.

En C# comme en Java, seul l'héritage simple est permis. Un mot clé unique, `base()`, permet de référer au constructeur choisi du parent.

La syntaxe d'initialisation du parent en C# suit celle de C++ : l'appel à `base()` précède l'accolade ouvrante du constructeur de l'enfant.

Toutefois, comme en Java, un appel de méthode virtuelle dans un parent sollicite la méthode de l'objet réellement en cours de construction. *En apparence, la mécanique de construction suit la démarche de C++, mais en pratique c'est la stratégie de Java qui s'applique*.

L'objet est créé complètement lors de l'invocation de `new`, incluant la mise en place des références aux méthodes virtuelles, et les constructeurs se limitent ensuite à initialiser les attributs.

Notez que C# offre des destructeurs, et que les méthodes virtuelles y invoqueront aussi les versions des méthodes de l'objet effectivement détruit, mais que la prudence reste de rigueur.

```
using System;
namespace z
{
   class X
   {
      public X()
      {
         f();
      }
      public virtual void f()
      {
         Console.WriteLine("X.f()");
      }
   }
   class Y : X
   {
      public Y()
         : base()
      {
      }
      public override void f()
      {
         Console.WriteLine("Y.f()");
      }
      static void Main(string[] args)
      {
         Y y = new Y();
      }
   }
}
```

En VB.NET, la situation se rapproche de celle rencontrée en C# mais avec quelques nuances syntaxiques.

Une clase `VB.NET` est abstraite si elle est qualifiée **`MustInherit`**. Une méthode `VB.NET` est abstraite si elle est qualifiée **`MustOverride`**.

Comme en C++, Java ou C#, une classe doit être abstraite si elle comprend au moins une méthode abstraite.

Les particularités quant au polymorphisme en C# et en `VB.NET` sont les mêmes, aux mots-clés près (et au fait qu'en `VB.NET`, `MyBase.New()` doit être invoqué entre le `Begin Sub` et le `End Sub` de la méthode `New` de l'enfant).

```
Namespace z
   MustInherit Class Forme
      ' bla
      public MustOverride Sub Dessiner()
   End Class
   Class Carré
      Inherits Forme
      Private Const HAUTEUR As Integer = 5
      Private Const LARGEUR As Integer = HAUTEUR
      ' bla
      Public Overrides Sub Dessiner()
         For i As Integer = 0 To HAUTEUR - 1
            For j As Integer = 0 To LARGEUR - 1
               System.Console.Write("*")
            Next
            System.Console.WriteLine()
         Next
      End Sub
   End Class
   Public Class Z
      Public Shared Sub Main()
         Dim c As Carré = New Carré
         c.dessiner()
      End Sub
   End Class
End namespace
```

Exercices – Série 04

EX00 – Faites en sorte que le programme suivant affiche correctement un carré par défaut, puis un triangle par défaut et un rectangle par défaut.

Cet exercice présume que vous avez rédigé les classes en question. ***Ne modifiez pas la moindre ligne du programme à droite!***

```
#include "Forme.h"
#include "Carre.h"
#include "Triangle.h"
#include "Rectangle.h"
int main() {
   const int NB_FORMES = 3;
   Forme *p[NB_FORMES] = {
      new Carre, new Triangle, new Rectangle
   };
   for (int i = 0; i < NB_FORMES; i++) {
      p[i]->dessiner(); // polymorphisme!
      delete p[i];
   }
}
```

EX01 – Inspirez-vous du programme proposé à l'exercice EX00 et modifiez-le pour que (a) l'usager choisisse le nombre de formes à dessiner; (b) pour chaque forme, l'usager choisisse le type de forme (une parmi les trois ci-dessus); et (c) une fois toutes les formes choisies (et pas avant!) chaque forme soit affichée, dans l'ordre.

EX02 – Inspirez-vous du programme proposé à l'exercice EX01 et modifiez-le pour que (a) l'usager choisisse le nombre de formes à dessiner; (b) pour chaque forme, l'usager choisisse le type de forme (une parmi les trois ci-dessus) et les dimensions de la forme (les questions à poser varieront selon la forme); et pour que (c) une fois toutes les formes choisies (et pas avant!), chaque forme soit affichée, dans l'ordre.

EX03 – Réalisez le travail proposé à ***Exercices – Série 10*** de [POOv00]. Apportez les modifications suivantes à la structure de votre programme :

- faites en sorte que la classe Joueur soit une classe abstraite, à cause de la méthode abstraite GetPoints() retournant un entier;
- créez trois classes dérivées de la classe Joueur, respectivement nommées Avant, Defenseur et Gardien;
- la méthode GetPoints() d'un Avant retournera les points de ce joueur, comptant un point par passe et deux points par but;
- la méthode GetPoints() d'un Defenseur retournera les points de ce joueur, comptant un point par passe et deux points par but – ce sont les mêmes règles que pour un Avant, mais avoir deux classes distinctes nous permettra d'inclure éventuellement d'autres facteurs, comme le temps passé sur la glace par exemple;
- la méthode GetPoints() d'un Gardien retournera les points de ce joueur, comptant deux points par victoire, un point par match nul, et un bonus de trois points par blanchissage;
- le programme devra faire en sorte que chaque équipe ait un ou deux gardiens, et entre quatre et six défenseurs;
- vous devrez modifier votre programme pour qu'il tienne compte de tous ces facteurs. Il y a un certain nombre de modifications requises qui ne sont pas explicitement mentionnées ci-dessus

(entre autres, les attributs requis pour gérer les points d'un `Gardien` sont différents de ceux requis pour gérer un `Avant` ou un `Defenseur`), alors il sera à votre avantage de réfléchir au problème sur papier avant d'essayer d'implanter une solution;

- vous pouvez ajouter des classes si vous pensez que ça peut vous aider.

Note :	votre tâche sera grandement simplifiée si vous implantez une classe `Equipe`, représentant une équipe complète, ceci vous permettant d'implanter les règles de gestion d'équipe à même un objet cohérent. Si vous procédez ainsi, vous serez étonné(e) de voir à quel point il deviendra simple d'implanter un gestionnaire tout entier de pool de hockey!

Encadrer le polymorphisme et l'abstraction

Les usages des programmeuses et des programmeurs `OO` évoluent, et certaines pratiques ont fini par prendre racine dans divers cercles. Parmi ceux-ci, on retrouve une préoccupation de fond : comment bien gérer la mise en place de comportement polymorphiques dans une hiérarchie?

Rappelons que, pour une classe dérivée, le polymorphisme est la ***capacité*** de spécialiser le comportement d'un parent, alors que l'abstraction est l'***obligation*** de le spécialiser.

Pour le code client, utiliser indirectement une entité polymorphique implique à la fois l'existence d'un comportement sujet à spécialisation et la certitude que l'invocation du comportement résultera en l'exécution de la forme la plus spécialisée possible.

⇒ Pour un client, le polymorphisme est une ***garantie***.

> Si `Presentable` expose une méthode polymorphique `presenter()` qui n'est pas abstraite, donc avec une implémentation par défaut, alors tout `Presentable` saura se présenter au moins comme un `Presentable` et peut-être de manière plus spécifique encore. Si `Dessinable` expose la méthode abstraite `dessiner()`, alors tout `Dessinable` devra spécialiser `dessiner()`.
>
> Le code client sait qu'un `Dessinable` implémente une méthode `dessiner()` et que tout `Presentable` saura se présenter.

Il arrive par contre qu'on souhaite encadrer le recours au polymorphisme, de manière à laisser les enfants spécialiser les comportements des parents, soit, mais sans nécessairement les laisser complètement libres de faire n'importe quoi.

Une manière de permettre aux enfants de standardiser volontairement certains comportements polymorphiques est d'implémenter certains services au niveau du parent. L'approche présentée dans la section ***Méthodes abstraites implémentées***, plus haut, en est un exemple : si les enfants le souhaitent ils peuvent déléguer, en tout ou en partie, leur comportement polymorphique à une version standardisée implémentée par le parent. Le polymorphisme sans méthodes abstraites est semblable, en ce sens que si l'enfant ne spécialise pas le comportement du parent, alors le comportement par défaut (du parent) s'applique tout simplement; avec des méthodes abstraites implémentées, cette mécanique devient tout simplement un choix explicite chez l'enfant.

Méthodes modèles (idiome NVI)

Une autre approche possible est de laisser le parent encadrer le comportement polymorphique des enfants, que ce soit pour assurer l'exécution systématique de code par défaut avant ou après l'appel polymorphique, réaliser des traces, de la comptabilité, gérer des considérations de synchronisation, assurer le respect de préconditions ou de garanties de postconditions, ou pour toute autre raison. Cet encadrement délibéré des enfants par le parent est plus souvent souhaitable qu'on pourrait le penser.

Remarquez ici une dichotomie entre l'*encadrement*, choisi par le parent, donc prédéterminé et typiquement non polymorphique, et le *comportement encadré*, qui est polymorphique (peut-être même abstrait) et qui est destiné à être spécialisé au moins sur une base occasionnelle. Une pratique connue, le recours aux **méthodes modèles**, ou *Template Methods*[62], se prête très bien à cette tâche.

> **Herb Sutter** nomme[63] la version idiomatique proposée ici l'**idiome NVI**, pour *Non-Virtual Interface*. Personnellement, j'aime bien le terme français ***devanture***, mais puisqu'il existe un schéma de conception reconnu du nom de façade, ce terme porte parfois à confusion…

Pour appliquer cette approche :

- il faut s'assurer que toute méthode virtuelle, outre les destructeurs, soit protégée (pas publique);
- il faut offrir, à même la classe parent, une méthode non virtuelle publique qui invoque, à l'interne, la méthode virtuelle protégée;
- le code client invoquera le comportement polymorphique à partir de la méthode non polymorphique;
- les enfants spécialiseront le comportement polymorphique comme à l'habitude.

```
// Parent.h
class Parent {
protected:
   // abstraite ou non,
   // selon les besoins
   virtual void f_impl() = 0;
public:
   // non polymorphique
   void f() {
      // ... avant
      f_impl();
      // ... après
   }
   virtual ~Parent() = default;
};
// Enfant.h
class Enfant : public Parent {
private:
   void f_impl() override {
      // ...
   }
};
```

Le passage forcé par une méthode du parent permet de contrôler avec précision les opérations avant et après l'invocation polymorphique, ce qui rejoint directement nos objectifs.

L'illustration à droite donne un exemple d'application de cette approche. `Enfant` dérive de `Parent` et spécialise le comportement polymorphique `f_impl()`. De son côté, `Parent` qualifie `f_impl()` protégée et expose le service `f()`, non polymorphique, au code client. La méthode `f()` aura un nom utile pour le code client et encadrera `f_impl()`.

Notez que l'une des qualités de cette approche est qu'*elle peut être appliquée systématiquement, qu'il y ait ou non encadrement*. En effet, si la méthode publique ne fait que relayer l'appel à la méthode protégée, le compilateur peut sans peine optimiser le tout; de plus, si le besoin d'encadrement survient ultérieurement, il sera plus facile à ajouter si l'idiome est déjà mis en application.

---

[62] Il ne faut pas confondre le mot *Template* dans le nom de cette stratégie avec le mot technique `template` du langage C++. Les *templates* de C++ [POOv02] constituent un métalangage statique permettant la description de types et de comportements génériques, alors que le *Template Method* est une technique par laquelle une devanture isole et encadre un comportement polymorphique.

[63] Voir [SutterNvi]. L'idiome `NVI` peut aussi être vu comme une version pointue du schéma de conception Stratégie (voir [EffCpp], Item 35, par **Scott Meyers**, et en général le chapitre 6 de ce livre, pour une discussion en profondeur).

Héritage et paramètres par défaut

Il existe un danger en apparence mineur et somme toute technique à combiner héritage, polymorphisme et méthodes munies de paramètres avec valeurs par défaut. À vrai dire, ce mélange est à éviter. Voici pourquoi.

Prenons pour exemple le programme proposé ci-dessous (rappel : un `struct` équivaut à une `class` à ceci près que, par défaut, les membres sont publics et l'héritage est public).

Le programme principal y instancie successivement un `X` et un `Y`, les deux à travers un `X*`, puis invoque chaque fois la méthode polymorphique `f()`. Sans surprises, le polymorphisme s'applique, donc le premier appel invoque `X::f()` alors que le second invoque `Y::f()`.

Ce qui peut surprendre est la valeur du deuxième paramètre lors de l'invocation de `f()`. En effet, les concepteurs de la classe `X` ont apposé une valeur par défaut (la valeur `3`) au paramètre `val` de la méthode `f()`, alors que les concepteurs de la classe `Y`, eux, y ont apposé la valeur par défaut `4`. Dans un cas comme dans l'autre, nous sommes en droit de supposer que les concepteurs ont choisi ces valeurs avec soin.

Cependant, une exécution du programme principal de cet exemple montrera clairement que, si l'invocation de `f()` est (tel qu'indiqué plus haut) bel et bien polymorphique, *la valeur par défaut du paramètre `val` passé à la méthode `f()` sera ici `3` dans chaque cas*.

> Considérant les choix de design faits par les concepteurs de ces deux classes, ce choix risque de les surprendre, et constitue probablement, dans les circonstances, un bug sournois.

```
#include <iostream>
using namespace std;
struct X {
   virtual void f(ostream &os, int val = 3) {
      os << "X::f(" << val << ")" << endl;
   }
   virtual ~X() = default;
};
struct Y : X {
   void f(ostream &os, int val = 4) override {
      os << "Y::f(" << val << ")" << endl;
   }
};
int main() {
   apX = new X;
   pX->f(cout); // X::f(cout,3)
   delete pX;
   pX = new Y;
   pX->f(cout); // Y::f(cout,3)
   delete pX;
}
```

La raison de ce résultat est que l'invocation polymorphique est résolue *à l'exécution*, à partir du type de l'objet réellement accédé, alors que l'attribution de valeurs par défaut à des paramètres est résolue *à la compilation*, les valeurs de remplacement dépendant des types connus alors.

Ici, le résultat est un bug logique : les attentes des concepteurs des classes dérivées ne peuvent être rencontrées. Spécialiser les valeurs par défaut des paramètres à des méthodes polymorphiques est un signe de design fautif ou d'une incompréhension de la mécanique.

Héritage et classes concrètes

Le polymorphisme est un élément de design, pas un accident. Un comportement polymorphique repose sur une façon de faire qui a été pensée à même une classe parent et spécialisée par des classes enfants; si la classe parent n'a pas été pensée de manière à favoriser le polymorphisme, alors il est sage de ne pas en dériver.

En fait, tenter de dériver publiquement d'un parent non polymorphique est *dangereux*. L'héritage public exprime l'enfant comme une spécialisation comportementale du parent, dans le but éventuel d'utiliser un pointeur ou une référence sur le parent menant en fait vers l'enfant. Les classes non polymorphiques ne sont pas prêtes à jouer ce rôle.

Pour illustrer les risques de dériver publiquement d'un parent non polymorphique, prenons les classes `Entier` et `Note` (à droite). La classe `Entier` n'est pas polymorphique et n'a pas été prévue pour servir de devanture pour une classe dérivée. Par conséquent, un programme comme celui proposé à droite entraînera des fuites de ressources : le constructeur de `Note` est invoqué dans le programme principal mais le destructeur de la classe `Entier` n'est *pas* virtuel (et n'a aucune raison de l'être!); l'invocation de `delete p;` appellera non pas le destructeur de `Note` mais bien celui d'`Entier`.

Dériver un enfant d'un parent qui n'a pas pris les dispositions pour servir d'abstraction pour d'autres est une erreur dont les conséquences peuvent être graves. De toute manière, puisque la classe `Entier` n'est pas polymorphique, la variable `p` (dans `main()`) ne donnera accès qu'aux méthodes de `Entier`, pas à celles de `Note`.

Utiliser `p` comme une spécialisation d'`Entier` ne comporte que des inconvénients. Cet exemple peut sembler bête, puisqu'il repose sur l'idée que l'erreur serait d'utiliser à travers une abstraction qui ne s'y prête pas (`Entier`) un type pour lequel les bénéfices apparents de cette manœuvre sont faibles ou nuls. Plusieurs s'y font prendre malgré tout sans s'en apercevoir dans ces cas qui semblent *a priori* mieux pensés.

```
class Entier {
   int val_;
public:
   Entier(int val = 0) : val_{val} {
   }
   int valeur() const noexcept {
      return val_;
   }
};
class Note : public Entier {
public:
   Note(int val) : Entier{val} {
      // ouvrir une BD
   }
   // ...
   ~Note() {
      // fermer la BD
   }
};
int main() {
   Entier *p = new Note;
   delete p; // oups!
}
```

Prenons par exemple cette tentative (bien intentionnée, mal réalisée) d'exprimer une chaîne de caractères dont la taille maximale, exprimée en nombre de caractères, ne peut dépasser `100`.

Ici, les concepteurs ont pensés bien faire en dérivant de `std::string` et en surchargeant `push_back()` de manière à dissimuler la version du parent et à lever une exception si la capacité est susceptible d'être dépassée suite à une opération, par exemple l'application de la fonction `f()` sur une `tite_chaine()`.

Imaginons que la manière par laquelle s'exprime cette chaîne de caractères soit par héritage simple où la classe `tite_chaine` est l'enfant et `std::string` le parent. Y a-t-il un risque?

Absolument. En effet, le contrat pour `std::string` et sa méthode `push_back()` est différent du contrat de `push_back()` pour `tite_chaine`, ce qui implique que la fonction `f()` appliquée à une `tite_chaine` représente un bris de contrat : il est possible de considérer l'enfant comme un cas particulier du parent mais le parent et l'enfant ne partagent pas les mêmes règles de validité pour ce qui est de l'opération `push_back()`. La fonction `f()` est une illustration claire de ce danger : il suffit de traiter l'enfant comme son parent pour tout casser.

Appliquer l'héritage public à un parent qui n'expose aucune méthode virtuelle est une erreur de design car l'enfant et le parent exposent tous deux des contrats sémantiquement distincts. Cette situation est partie intégrante de tout contrat manifeste entre parent public et enfant sans polymorphisme.

L'héritage (public!) est un outil précieux mais pas une panacée ou un marteau doré.

```
#include <string>
#include <iostream>
using namespace std;
class Pleine {};
class tite_chaine
   : public string {// pas gentil
   // ...
public:
   // ...
   void push_back(char c) {
      if (size() == 100)
         throw Pleine{};
      string::push_back(c);
   }
};
void f(string &s) {
   s.push_back('A');
}
int main() {
   tite_chaine s;
   for (int i = 0; i < 100; ++i)
      s.push_back('A');
   try {
      s.push_back('Z');
      cout << "Oups!";
   } catch (Pleine&) {
      cout << "Ok!";
   }
   f(s); // boum!
}
```

Exposer les membres d'un parent et clauses `using`

En C++, l'exposition dans un enfant d'un membre de l'un de ses parents se fait sur la base du nom du membre en question. Par exemple, examinez ce qui suit :

```
struct X {
   int f(int n) const {
      return n + 1;
   }
   int f(double d) const {
      return static_cast<int>(d);
   }
};
struct Y : X {
};
int main() {
   Y y;
   int n = y.f(3);
   n += y.f(3.5);
}
```

Les deux appels à `y.f()` dans `main()` compileront sans problème puisque `X` expose deux méthodes d'instance nommées `f()`, l'une prenant un `int` en paramètre et l'autre prenant plutôt un `double`.

Cependant, examinez maintenant ce qui suit :

```
// ... classe X, plus haut...
#include <string>
using std::string;
struct Z : X {
   int f(const string &s) const {
      return static_cast<int>(s.size());
   }
};
int main() {
   Y y;
   int n = y.f("J'aime mon prof"); // Ok
   // n += y.f(3.5); // ne compile pas!
}
```

Le fait de définir une méthode nommée `f` dans `Z` a pour impact de cacher les noms `f` de son parent `X`, sans égard à la signature des méthodes en question.

Il est possible pour l'enfant d'exposer de façon volontaire des noms d'un parent avec une clause `using`. Un exemple est proposé à droite.

Évidemment, si l'enfant déclare explicitement a une méthode de même signature que celle du parent (p. ex. : si `Z` déclare une méthode d'instance de signature `int f(int)` alors que `X` expose aussi une méthode de même signature), alors la méthode de l'enfant cache tout simplement celle du parent.

```
// ... classe X, plus haut...
#include <string>
using std::string;
struct Z : X {
   int f(const string &s) const {
      return static_cast<int>(s.size());
   }
   using X::f; // exposer X::f(int) et
               // X::f(double) à même Z
};
int main() {
   Y y;
   int n = y.f("J'aime mon prof"); // Ok
   n += y.f(3.5); // Ok
}
```

Clause `using` et constructeurs

Depuis C++ 11, il est possible pour un enfant d'exposer par une clause `using` des constructeurs de son parent (avec les classes ci-dessus : `using X::X`).

Exemple concret :

```
class B {
   int val_;
public:
   B(int val) : val_{val} {}
};
class D : public B {
   float f_;
public:
   D(float f, int val) : B{val}, f_{f} {
   }
   using B::B;
};
```

Ici, la clause `using` introduit dans `D` un constructeur `D::D(int)` qui appellera `B::B(int)` en lui relayant le paramètre reçu à la construction du `D`. Le membre restant de `D` (l'attribut `f_`) sera tel qu'il aurait été s'il n'avait pas été explicitement initialisé par `D` :

```
// ceci sera introduit par le compilateur
D::D(int nom_inconnu) : B(nom_inconnu) {}
```

Si les attributs résiduels n'ont pas de constructeur par défaut (p. ex. : le `float` nommé `f_` dans `D`), alors ils demeureront non-initialisés… Attention au respect de l'encapsulation!

L'héritage comme politique

L'approche `OO` est une approche qui a beaucoup évolué, mais pour laquelle les gens imaginatifs trouvent encore aujourd'hui des usages insoupçonnés. En particulier lorsqu'elle est couplée avec des pratiques de programmation générique [POOv02].

Entre autres choses, le point de vue que nous avons de l'héritage a beaucoup évolué au fil des ans. Voyons quelques points de vue sur la chose, dont certains diffèrent des points de vue véhiculés sur le sujet à l'origine.

Hériter pour ajouter

L'idée originale derrière l'héritage était que la classe `D` dérive de la classe `B` si un `D` est en fait un `B` auquel on aurait ajouté quelque chose (héritage d'implémentation). Suivant cette approche quelque peu cumulative, plus l'enfant possède de niveaux d'ancêtres et plus grande sera la consommation d'espace mémoire de l'enfant.

L'idée selon laquelle *hériter implique réutiliser* reposait à l'origine sur cette vision des choses : être tout ce que le parent est et même plus permettait de ne pas avoir à réécrire le code du parent. En retour, n'utiliser que ce point de vue tend à résulter en des hiérarchies dont les objets sont lourds et accumulent le bois mort; chaque enfant étant au moins aussi gros que son parent, pour appliquer une terminologie inspirée de l'héritage simple, plus les objets sont profonds dans une hiérarchie donnée et plus ils sont gros et lourds.

Comme dans toute chose, il ne faut pas jeter le bébé avec l'eau du bain. L'idée a du bon, mais il faut s'en servir avec discernement.

Hériter pour spécialiser

Le polymorphisme introduit l'approche selon laquelle *hériter signifie spécialiser*. Si la classe `B` expose une méthode virtuelle, à plus forte partie si cette méthode est abstraite, alors faire de la classe `D` un dérivé de `B` implique qu'on permette à `D` de spécialiser certains comportements de `B` (ou, dans le cas où `B` est abstraite, qu'on oblige `D` à spécialiser certains comportements de `B`) Cela implique aussi qu'on souhaite, dans les programmes, utiliser `B` à titre d'abstraction pour `D`, donc utiliser un `B` sans savoir s'il s'agit ou non d'un `D`.

Cette stratégie a mené, dans les langages à héritage simple comme Java et C#, aux interfaces en tant que concept du langage, et à certaines réactions épidermiques selon lesquelles on devrait se débarrasser de l'héritage d'implémentation et ne conserver que l'héritage d'interfaces[64].

Comme dans toutes les positions dogmatiques, il vaut la peine de respirer quelques instants et de se souvenir que la beauté est dans la modération. On tend trop souvent, lorsqu'une pratique est utilisée de manière abusive, à condamner la pratique plutôt que les abus.

---

[64] Voir entre autres [GosIntv], qui fait partie d'une longue entrevue avec ***James Gosling***, concepteur de Java.

Hériter pour contraindre

Une pratique très intéressante et qu'on connaît mieux en C++ que dans d'autres langages est celle d'hériter pour contraindre. Parfois, si la classe `D` dérive de la classe `B`, cela signifie que `D` est contraint à respecter les règles imposées par `B`.

Imaginons par exemple un type pour lequel on voudrait rendre impossible la copie (pratique à laquelle nous avons eu recours à quelques reprises dans [POOv00]). Il est assez simple d'empêcher la copie d'un objet : il suffit de déclarer son constructeur par copie et son opérateur d'affectation tous deux privés et de ne pas les définir.

Cela dit, cette technique doit être répétée pour toute classe dont les instances doivent être *incopiables*. Pourtant, l'idée d'*incopiable* est claire et définie… on peut même la nommer! Alors pourquoi ne pas lui donner forme de manière pleine et entière entant que type?

Idiome de la classe `Incopiable`

Imaginons que la classe `Incopiable` se décline comme suit.

Clairement, toute instance d'`Incopiable` ne pourra être passée par valeur à un sous-programme, et on ne pourra affecter un `Incopiable` à un autre `Incopiable`. Le concept est là, tout entier. Il porte un nom clair; il est à la fois code et documentation.

Le constructeur de copie et l'opérateur d'affectation sont publics pour faciliter la génération de messages d'erreur pertinents.

Avant l'avènement de la qualification `= delete`, la pratique était différente.

```
struct Incopiable {
   Incopiable(const Incopiable&) = delete;
   Incopiable&
      operator=(const Incopiable&) = delete;
protected:
   Incopiable() = default;
   ~Incopiable() = default;
};
```

Connaissant la pratique pour rendre des objets incopiables, on voudra l'utiliser comme telle dans les programmes. En retour, on voudra le caractère `Incopiable` d'un objet soit un détail d'implémentation[65], donc l'héritage public est à proscrire.

La clé est la suivante : une classe sera `Incopiable` si elle dérive de la classe `Incopiable`. La relation s'exprime au sens du verbe être, et se prête pleinement à l'héritage, mais à l'héritage privé.

Le parent sert ici à titre de politique applicable à ses enfants.

```
class X : Incopiable {
   // ...
};
```

[65] Souvenons-nous que l'héritage public a un sens précis : on veut utiliser les enfants de manière polymorphique, à partir des services du parent. Ici, à quoi cela servirait-il de faire un tableau de pointeurs sur des `Incopiable`, par exemple, puisque cette classe n'expose aucune méthode virtuelle?

La réutilisation de code est claire, le coût en espace est presque nul (ou même nul tout court; voir ***Héritage et consommation d'espace***), et l'intention est claire, le tout sans compromettre l'encapsulation. Le recours à un type dont le nom est clair résulte en un programme autodocumenté, dont l'intention est évidente.

Si l'héritage privé ne vous convient pas, une approche alternative pour rendre une classe incopiable est d'y déclarer un attribut d'instance `Incopiable`.

Cette approche mène au même résultat en pratique : les opérations de copie générées par le compilateur deviennent illégales (la construction par copie automatique invoque la construction par copie attribut par attribut, or celle-ci est impossible, et l'impact sur l'affectation est semblable). Elle est toutefois moins élégante.

```
class X {
   Incopiable bidon_;
   // ...
};
```

Cette perte d'élégance tient à (au moins) trois facteurs :

- la relation passe du verbe être (les instances de la classe *sont* incopiables), dont elle bénéficiait en vertu de l'héritage privé, au verbe avoir (les instances de la classe possèdent quelque chose d'incopiable), en vertu de la composition;
- il faut inventer un nom d'attribut bidon pour l'attribut `Incopiable` utilisé, alors que cet attribut n'a d'autre utilité que d'exister. Ce simple irritant peut nuire à la mise en application de la technique (l'inélégance rebute, c'est connu); et
- l'optimisation basée sur les parents vides [hdEBCO] n'est pas applicable quand une classe possède des attributs qui sont eux-mêmes des objets vides. Conséquemment, le compilateur possède moins d'outils pour réduire la taille des programmes avec la stratégie par composition qu'il n'en avait avec la stratégie par héritage.

Hériter pour contraindre est donc une manière intéressante d'indiquer, en vertu de la nature-même d'une classe (en vertu de ce qu'elle est), certaines contraintes et certaines politiques s'appliquant à elle.

Cette vision de l'héritage est encore plus importante lorsqu'on la combine avec des stratégies de programmation générique. Pour des idées, voir [hdTBN], [hdSelPar] et [hdEnPar].

Fonctions supprimées ou par défaut

Détail qui peut sembler anodin : le compilateur, lorsqu'il génère une fonction pour nous, tend à faire un meilleur travail que nous ne le ferions nous-mêmes. Par exemple :

A (version « manuelle »)	A (version « automatique »)

`class A {`
`   // ...`
`public:`
`   A();`
`   // ...`
`};`
	`class A {`
	`   // ...`
	`};`

Ici, le constructeur par défaut de la version automatique de `A` sollicitera implicitement les constructeurs par défaut des attributs de la classe `A`, mais le fera plus efficacement que ne le fait la version manuelle de `A`. Le compilateur sait ce qu'il fait; il connaît le contexte et agit en connaissance de cause; quand nous suppléons un constructeur par défaut vide, comme le fait par exemple la version « manuelle » de la classe `A` ci-dessus, cela introduit une incertitude, un « doute » : pourquoi donc la programmeuse ou le programmeur fait-elle/ fait-il cela?

La même situation peut survenir avec n'importe quelle fonction, en particulier celles qui constituent la Sainte-Trinité (constructeur de copie, destructeur, affectation) pour lesquelles le compilateur génère systématiquement des implémentations, à moins que nous ne les bloquions volontairement.

Il se trouve qu'il y a plusieurs raisons pour souhaiter éliminer une telle fonction, ou encore l'expliciter tout en souhaitant un comportement tel que celui que le compilateur aurait généré spontanément. Par exemple :

- souhaiter changer la qualification de sécurité par défaut (vouloir un destructeur privé ou protégé, par exemple);
- souhaiter un destructeur par défaut mais virtuel, pour fins de polymorphisme;
- souhaiter un objet incopiable, évidemment (supprimer la copie par affectation et par construction, mais souhaiter une construction par défaut et une destruction protégées); etc.

La solution est simple : depuis C++ 11, il est possible d'exprimer au compilateur que nous souhaitons une implémentation par défaut d'une fonction, tout comme il est possible de lui indiquer que nous souhaitons supprimer une implémentation qui, autrement, aurait été rendue disponible. L'écriture est simple : suffixer la signature de la fonction d'un **`= default`** pour obtenir le comportement par défaut, ou d'un **`= delete`** pour supprimer l'implémentation qui, dans d'autres circonstances, aurait été générée.

L'écriture contemporaine de la classe `Incopiable` profite directement de ces mécanismes :

```
struct Incopiable {
   Incopiable(const Incopiable&) = delete;
   void operator=(const Incopiable&) = delete;
protected:
   Incopiable() = default;
   ~Incopiable() = default;
};
```

Ici, le compilateur est informé de l'intention de la programmeuse ou du programmeur : « je souhaite éliminer les opérations de copie, et je veux une implémentation par défaut mais protégée du constructeur par défaut et du destructeur ». Le code sera au pire aussi bon qu'auparavant, probablement bien meilleur, et le résultat sera le même.

Il existe des raisons plus générales de vouloir supprimer des fonctions. Pensez par exemple à une fonction ayant la signature suivante :

```
float f(float);
```

S'il s'avère que la tâche accomplie par `f()` est associée de près au type `float`, et poserait problème si on lui passait un `int` ou un `double`, alors il est possible d'indiquer au compilateur que les versions acceptant un `int` ou un `double` ne sont pas souhaitées :

```
float f(float);
int f(int) = delete;
double f(double) = delete;
```

Ainsi, un appel à `f(3)` ou à `f(3.0)` ne compilera pas, mais un appel à `f(3.0f)` compilera sans peine.

Dans d'autres langages

Les méthodes modèles peuvent être rédigées dans d'autres langages que C++, mais en portant attention à certains détails.

En Java, la stratégie appliquée est la même qu'en C++, à ceci près que nous devons empêcher la spécialisation du service public `f()` du fait que, par défaut, cette méthode sera polymorphique.

Notez que Java interdit le raffinement de la sécurité d'une méthode lors de sa spécialisation. Ici, `f_impl()` dans `Enfant` peut demeurer protégée (notre choix ici) ou devenir publique, mais ne peut pas être plus stricte que chez `Parent` (on ne peut la rendre privée).

```
abstract class Parent {
   protected abstract void f_impl();
   public final void f() {
      // ... avant
      f_impl();
      // ... après
   }
}
class Enfant extends Parent {
   protected void f_impl() {
      // ...
   }
}
```

En C#, comme en C++, une méthode n'est considérée polymorphique que si elle a été déclarée `virtual` à l'origine. Pour cette raison, l'extrait à droite est très proche de l'exemple C++ exploré plus haut.

Comme dans le cas des exemples Java et C++, nous avons une spécialisation privée de `f_impl()` dans `Enfant` et la version de `f_impl()` déclarée dans `Parent` est abstraite. Ce sont des choix esthétiques, pas des contraintes techniques.

```
namespace z
{
   abstract class Parent
   {
      protected abstract void f_impl();
      public void f()
      {
         // ... avant
         f_impl();
         // ... après
      }
   }
   class Enfant : Parent
   {
      private void f_impl()
      {
         // ...
      }
   }
}
```

En VB.NET, la situation est légèrement différente :

- le service à spécialiser demeure protégé (et est abstrait dans notre exemple, bien qu'il eut pu être simplement polymorphique sans que cela n'ait un impact négatif sur notre démarche);
- l'encadrement demeure public; mais
- comme en Java, il est illégal, en VB.NET, de spécialiser une méthode d'instance en raffinant son niveau de sécurité. Ici, donc, la méthode f() de la classe Enfant doit être protégée comme l'était la méthode f() de la classe Parent. Ceci explique la différence de qualification entre cet exemple et les autres.

Pour le reste, par contre, la mécanique est la même.

```
Module z
   Public MustInherit Class Parent
      Protected MustOverride Sub f_impl()
      Public Sub f()
         ' ...avant
         f_impl()
         ' ...après
      End Sub
   End Class
   Public Class Enfant
      Inherits Parent
      Protected Overrides Sub f_impl()
         ' ...
      End Sub
   End Class
End Module
```

Conversions explicites de types

Pourquoi cette section?

La présence d'héritage implique fréquemment qu'on traite un enfant comme l'un de ses parents, par souci de généralisation. L'inverse est aussi parfois vrai, mais mène à des risques particuliers et tient souvent de fautes de design. Comprendre les enjeux de cette section aide à avoir une meilleure prise sur les points d'intérêt de la conception de systèmes OO.

Il peut arriver qu'on veuille explicitement traiter une variable ou une constante d'un type donné comme s'il s'agissait d'une variable ou d'une constante d'un autre type, ou encore (ce qui nous intéresse particulièrement) qu'on désire traiter une instance d'une classe comme une instance d'une autre.

Parfois, cette conversion est implicite et évidente; à d'autres moments, il s'agit d'une tâche périlleuse. Parfois, il est possible pour le compilateur de vérifier, au moment de la compilation, si la conversion est légale ou non; dans d'autres cas, il faut que le moteur du langage infère dynamiquement (à l'exécution) la validité de la tentative de conversion.

Les langages C, C++ et Java partagent la syntaxe C du transtypage, ou conversion explicite de type (en anglais *typecasts*, ou simplement *casts*), qui est de préfixer l'expression à convertir du nom du type dans lequel la conversion doit être réalisée, en mettant ce type entre parenthèses.

Par exemple :

```
int val_entiere = 3;
float val_reelle = 3.0f;
bool val_booleenne = false;
// l'opération qui suit prend la valeur de val_booleenne, crée une variable
// temporaire de type float ayant la valeur équivalente à false (ce qui va
// d'ailleurs donner 0.0f, pour des raisons techniques et historiques), puis
// affectera cette valeur temporaire à la variable val_reelle
val_reelle = (float) val_booleenne; // ou float (val_booleenne)
// l'opération qui suit prend la valeur de val_entiere, crée une variable
// temporaire de type bool ayant la valeur équivalente à 3 (ce qui va d'ailleurs
// donner true, pour des raisons techniques et historiques, mais lancera un
// avertissement assez particulier), puis affectera cette valeur temporaire à la
// variable val_booleenne
val_booleenne= (bool) val_entiere; // ou bool (val_entiere)
```

Autre exemple, celui-ci avec des classes :

```
class B { /* ... */ };
class D : public B {
   // ...
};
int main() {
   B b;
   D d;
   B *pb = &b; // Ok
   pb = &d;    // Ok: un D est un B
   D *pd = &d; // Ok
   // Illégal : pb est un B*, peu importe ce vers quoi il pointe. La classe B peut
   // avoir des enfants qui ne sont pas des D, donc ceci serait TRÈS risqué
   pd = pb;
   // Si vous êtes vraiment certain(e) de pb pointe vers un D, vous pouvez forcer
   // le compilateur à accepter votre demande, mais prenez soin de documenter le tout
   // car cette opération est foncièrement dangereuse
   pd = (D*) pb; // à vos risques et périls
}
```

C++ permet aussi une conversion *format constructeur*, au sens où `A a=(A)b;` et `A a=A(b);` sont équivalents et affecteront tous deux à `a` la valeur de `b` convertie en `A`, si cette conversion est possible. Cette syntaxe met d'ailleurs en relief que `A(b)` génère un `A` temporaire et ne modifie pas `b`.

Tout transtypage est un mensonge. Comme dans la vie, il y a de petits mensonges sans grandes conséquences (des *Little White Lies*, comme le disent les anglais), par exemple cacher à quelqu'un les détails des préparatifs de son anniversaire et le mettant sur de fausses pistes, ou faire semblant qu'un `float` est en fait un `double`, et il y en a de plus sérieux, comme faire semblant qu'un pointeur sur un `int` est en fait un pointeur sur une `std::string` : imaginez les conséquences si le programme s'avisait par la suite d'en invoquer une méthode!

Peu importe sa gravité, le transtypage est un mensonge qu'il faut assumer. En mentant sur la nature des entités qu'il manipule, la programmeuse ou le programmeur contourne les règles du système de types du langage, et accepte de ce fait de réduire l'assistance que lui porte le compilateur en signalant les risques et les erreurs lorsqu'il les rencontre. Assumer ce choix de mentir au compilateur se fait par le recours à une opération explicite de conversion de types, en le documentant, et en se donnant les moyens de retrouver cette opération ultérieurement pour chercher, si possible, une solution qui ne reposerait pas sur un contournement des règles.

Problèmes du repérage de la conversion

Les conversions C ont quelques défauts. Entre autres :

- elles ne discriminent pas entre les types de conversion désirées, donc entre les diverses intentions que peuvent avoir les programmeuses et les programmeurs en demandant les conversions (on y reviendra sous peu); et
- elles sont difficiles, une fois insérées, à retrouver dans le code, ayant une apparence se situant entre une déclaration de variable et un prototype de sous-programme.

> Le système de types très riche de C++ offre beaucoup d'options aux programmeuses et aux programmeurs, ce qui explique l'importance pour ces derniers de pouvoir clarifier leur intention lors de conversions explicites de types.
>
> D'autres langage `OO`, comme Java et les langages `.NET`, offrent une gamme d'options beaucoup moins vaste, et se limitent (pour le meilleur et pour le pire) à des conversions semblables à celles du langage C (voir ***Dans d'autres langages***). Ceci explique pourquoi la présente section est technique et fortement orientée C++.

On remarquera que ce type d'opération a un petit côté malsain. En effet, la grande majorité des opérations de transtypage sont vouées à une durée de vie très brève (ou, du moins, devraient l'être) et nous devrions viser à les éliminer de nos programmes.

Dans bien des cas (pas tous), le recours au transtypage résulte d'une faute de design, du besoin de corriger des différences philosophiques un peu trop tard dans le développement d'un projet (par exemple lors de la fusion de bibliothèques qui n'ont pas été pensées de manière compatible), de l'urgence d'apposer un correctif rapide à un bout de code, ou même de la simple manifestation d'un fond de paresse de la part des développeurs (ça arrive).

On souhaite donc pouvoir, le temps venu, retrouver ces opérations et les remplacer par des opérations plus élégantes ou moins risquées. Que la syntaxe des opérations de transtypage C ressemble d'aussi près à celle des appels de fonctions les rend malheureusement difficiles à repérer; trop souvent, ces opérations s'incrustent.

Il reste que des conversions explicites de types sont parfois nécessaires, quand bien même ce ne serait que par souci de pragmatisme. Il faut donc un palliatif à ces irritants.

Note : ce qui suit ne fait pas l'apologie du transtypage; on essaiera de l'éviter le plus possible. Le langage C++ se veut pragmatique : le transtypage permet de contourner certaines fautes de design, et d'exploiter à travers un programme bien conçu des outils plus anciens. Prenez-le donc comme un outil auquel on doit parfois avoir recours, bien qu'à contrecœur, parce qu'on veut livrer un produit opérationnel à temps ou parce qu'on a frappé l'un de ces étranges cas où il s'avère nécessaire.

Opérateurs de transtypage ISO

La norme ISO du langage C++ vise à régler ces deux irritants en proposant de nouveaux opérateurs de transtypage. Pris ensembles, ces opérateurs permettent de faire tout ce que la conversion C fait, mais de manière plus contrôlée et moins dangereuse. Chacun de ces opérateurs est voué à une forme particulière de conversion.

Voir [ISOCast] pour une explication (prudence : ceci date de 1993 alors que le standard officiel date de 1998) des raisons derrière les opérateurs de transtypage ISO.

Opérateur	Rôle
const_cast	Passer temporairement outre une spécification const[66]. Sert entre autres à contourner des fautes de design ayant trait à l'absence de qualification const sur des méthodes qui auraient dû être qualifiées ainsi.
reinterpret_cast	Effectuer une conversion de nature *malpropre*[67], à l'aveugle, entre deux adresses (convertir un int* en char*, par exemple). Surtout utilisé lors de manœuvres impliquant de la programmation de bas niveau, près de la machine.
static_cast	Effectuer une conversion *naturelle*, pouvant être validée au moment de la compilation.
dynamic_cast	Effectuer une conversion OO *risquée*, dont on devra valider la faisabilité au moment de l'exécution. Permet de naviguer une hiérarchie de classes et de valider le succès ou l'échec de la tentative de conversion.

Nous mettrons l'accent dans cette section sur les opérateurs static_cast et dynamic_cast, intimement liés au transtypage entre classes apparentées l'une à l'autre à travers une relation d'héritage.

Nous discuterons de const_cast<> et de reinterpret_cast<> un peu plus loin car, bien qu'ils soient tous deux utiles, ils sont moins centraux aux thématiques OO de base.

Tous les opérateurs de transtypage sont utilisés selon la même syntaxe. L'exemple qui suit utilise static_cast entre deux types T et U dans le but de montrer la forme générale.

Pour convertir A du type T au type U et entreposer le résultat dans B (de type U) avec static_cast, on exprimera l'opération de la manière proposée à droite.

```
T A;
// ...
U B = static_cast<U>(A);
```

[66] …ou volatile. Le sujet des méthodes et des variables qualifiées volatile est une problématique liée à la fois au modèle OO et à la multiprogrammation.

[67] Imaginez qu'on ait char c, *pc; int *pi; donc un entier sur huit bits (la variable c), un pointeur de char (nommé pc) et un pointeur de int (nommé pi). Écrire pc = &c; signifie que pc pointe vers c, ce qui est propre et légal : une opération comme *pc = 'A' déposerait un entier sur huit bits (le caractère 'A') dans une variable ayant une capacité de huit bits elle aussi (notre c). Mais si on indique pi = &c; alors on procède à une opération *très* dangereuse, puisque subséquemment *pi = 2; déposerait l'entier 32 bits de valeur 2 dans l'endroit pointé par pi… qui n'a *pas* une capacité de 32 bits, puisqu'il s'agit de la variable c, dont la capacité n'est que de huit bits. Si on veut quand même procéder à une affectation aussi dangereuse que celle de &c à pi, peu importe la raison, alors il faut recourir à un reinterpret_cast.

Conversion à la compilation : `static_cast`

L'opérateur `static_cast` permet d'effectuer un transtypage *à la compilation*. Seules des conversions entre des types apparentés seront possibles à l'aide de cet opérateur, ce qui ne signifie pas que la conversion soit nécessairement légale. Il est en effet possible de faire des bêtises même avec `static_cast`, ce que nous verrons dans les exemples plus bas.

Ainsi, on utilisera cet opérateur :

- pour convertir des données de types primitifs entre eux (convertir un `int` en `float`; convertir un `double` en `short`; *etc.*), puisque les règles pour ces conversions sont définies à même le standard du langage; et (ce qui est plus amusant)
- pour réaliser une conversion sécuritaire entre des pointeurs ou des références vers des instances de classes entre lesquelles il existe un lien de parenté, donc pour passer de l'enfant vers le parent ou du parent vers l'enfant.

Convertir une indirection vers un enfant en une indirection vers l'un de ses ancêtres est une opération qui s'avère souvent *implicitement correcte*, pour laquelle on pourrait omettre le transtypage (il y a des cas, avec l'héritage multiple, où le transtypage demeure nécessaire ici). Un enfant connaît ses parents (et, avec l'héritage public ou protégé, cela demeure vrai de manière transitive); que le compilateur est donc en mesure de valider ces demandes de conversion.

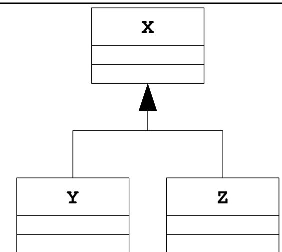
Convertir une indirection vers un ancêtre en une indirection vers un de ses descendants, par contre, est dangereux. Pour réaliser une telle manœuvre, le transtypage est ***obligatoire***. L'idée est que les programmeuses et les programmeurs, pour réaliser une opération risquée, doivent expliciter leurs intentions. Ceci équivaut à une prise de responsabilité de leur part.

Lorsque les programmeuses et les programmeurs ont la conviction que la conversion envisagée est légale, même s'il s'agit d'une conversion dangereuse en théorie, alors `static_cast` est une option de conversion valable. Cet opérateur modifie la perception qu'a le compilateur du type d'une donnée pour la durée d'une opération, et agit donc à la compilation seulement, souvent sans entraîner le moindre coût à l'exécution.

À titre d'illustration, examinons la hiérarchie suivante, composée de trois classes : un parent (`X`) et deux enfants publics directs (`Y` et `Z`).

L'exemple ci-dessous se base sur cette hiérarchie.

```
class X { /* ... */ };
class Y : public X { /* ... */ };
class Z : public X { /* ... */ };
```



L'exemple montre quelques applications de `static_cast`. Toutes les conversions y sont légales, bien que la dernière soit dangereuse. La légalité statique dépend des types connus à la compilation, mais la validité dynamique dépend des conditions lors de l'exécution.

```
int main () {
   Y y;
   X *px = static_cast<X *>(&y); // optionnel
   Y *py = static_cast<Y*>(px);  // Ok
   Z *pz = static_cast<Z*>(px);  // Ouch!
}
```

Question : pourquoi la dernière conversion est-elle légale mais dangereuse?

Légalité statique et validité dynamique

Examinons de plus près l'exemple ci-dessus pour comprendre chacune des trois expressions et, du même coup, mieux comprendre l'opérateur `static_cast`.

```
int main() {
   Y y;
   X *px = static_cast<X *>(&y); // optionnel
   Y *py = static_cast<Y*>(px);  // Ok
   Z *pz = static_cast<Z*>(px);  // Ouch!
}
```

Le commentaire apposé à la première conversion indique « optionnel ». La raison est que :

- tout `Y` est aussi un `X`, car `Y` dérive publiquement de `X`;
- tout membre public d'un `X` existe donc aussi pour un `Y`;
- il n'y a, clairement, aucun risque à traiter un `Y` comme un `X` : toute opération effectuée sur le `X` résultant mène nécessairement vers un membre valide. Ceci présume que les classes `Y` et `X` ont le même contrat sémantique (voir ***Polymorphisme***, plus haut, pour des détails).

Dans le cas où l'opération est visiblement légale et sans risque à la compilation, `static_cast` est le choix le plus approprié, ayant le coût le plus faible (souvent, ce coût sera nul) à l'exécution. Dans le cas où la conversion demandée est implicitement valide (c'est le cas ici), le recours à un `static_cast` peut simplement être omis.

La deuxième expression est un cas où le transtypage par `static_cast` est requis et sans danger, du moins aux yeux des programmeurs (le compilateur, lui, refuserait d'affecter `px` à `py` sans une demande de transtypage).

Voici pourquoi :

```
int main() {
   Y y;
   X *px = static_cast<X *>(&y); // optionnel
   Y *py = static_cast<Y*>(px);  // Ok
   Z *pz = static_cast<Z*>(px);  // Ouch!
}
```

- cette conversion vise à traiter un `X*` comme un `Y*`;
- par contre, tout `X` n'est pas nécessairement aussi un `Y`. En effet, un `X*` peut mener vers un `X` tout court, vers un autre dérivé de `X` (par exemple, un `Z`);
- ainsi, il est possible qu'il existe au moins un membre public d'un `Y` qui n'existe pas pour un `X`; donc
- il est dangereux de traiter un `X` comme un `Y`, toute opération effectuée sur le `Y` résultant pouvant mener vers un membre invalide.

Dans ce programme, nous voyons (d'un regard aérien) que px pointe en fait vers un `Y`, mais le compilateur, lui, ne voit que l'expression en cours et doit se baser sur les types impliqués. Pour lui, px mène vers au moins un `X`, ce qui ne permet pas de trancher quant au caractère raisonnable ou non de traiter ce pointé comme un `Y`. En appliquant un transtypage, la programmeuse ou le programmeur libère le compilateur de son dilemme : l'opération, risquée en général, est correcte dans ce cas particulier, et l'humain prend la responsabilité de cette interprétation.

La troisième est carrément dangereuse. Le pointeur px mène en fait vers un Y mais le programme ment et fait comme s'il pointait au moins vers un Z, ce qui est faux.

Ici, à partir de pz, le programme pourrait invoquer des méthodes de Z, qu'un Y ne possède pas, avec des résultats désastreux.

```
int main() {
   Y y;
   X *px = static_cast<X*>(&y); // optionnel
   Y *py = static_cast<Y*>(px);  // Ok
   Z *pz = static_cast<Z*>(px);  // Ouch!
}
```

L'expression Z *pz = px; était illégale pour une raison : il est impossible d'en garantir la validité en général, px pouvant mener vers une instance de n'importe quel type qui soit au moins un X. Dans la deuxième expression, le transtypage menait vers un résultat légal grâce à un savoir humain.

Dans la troisième, l'humain s'est trompé, mais le transtypage le rend responsable de sa propre erreur : le compilateur, lui, n'était pas d'accord, mais bon, si l'humain est tellement convaincu qu'il accepte de mettre sa tête sur le billot…

N'allez pas croire que static_cast permet n'importe quelle forme de mensonge aux programmeurs. Si nous ajoutons une quatrième ligne, où on tenterait de convertir un Y* (l'adresse de l'instance y) en Z* (le type de pz), le compilateur refuserait (avec raison), même avec un static_cast.

```
int main() {
   Y y;
   X *px = static_cast<X *>(&y); // optionnel
   Y *py = static_cast<Y*>(px);  // Ok
   Z *pz = static_cast<Z*>(px);  // Ouch!
   pz = static_cast<Z*>(&y);  // ILLÉGAL!
}
```

Ce refus tient au fait qu'il n'y a pas de parenté linéaire entre les types Y et Z (l'un n'est pas l'ancêtre de l'autre).

Si une conversion doit prendre deux indirections vers des types indirectement apparentés, comme dans le cas des types Y et Z ici, static_cast n'est pas la solution, et pour cause : le travail à accomplir pour en arriver à une conversion correcte, en général, est tout sauf banal.

En bref

Un static_cast, lorsque possible, est la meilleure option. Le résultat peut être obtenu à la compilation, et est typiquement soit un changement de perspective pour le compilateur, soit une addition d'une valeur constante sur une adresse.

Conversion à l'exécution : `dynamic_cast`

Contrairement à `static_cast`, qui repose (comme son nom l'indique) sur de l'information connue à la compilation des programmes, l'opérateur `dynamic_cast` repose (encore une fois, comme son nom l'indique) sur de l'information connue à l'exécution des programme. Cela fait du transtypage par `dynamic_cast` une option plus robuste, dans bien des cas, mais aussi plus lente qu'un `static_cast`.

⇒ On préférera un `static_cast` à un `dynamic_cast` quand une conversion *nécessairement valide* est requise parce qu'il s'agit de la conversion la plus efficace possible, étant réalisée à la compilation.

Quand le besoin surgit d'une conversion entre types apparentés (directement ou indirectement) et qu'un risque potentiel demeure, par exemple dans le cas où l'on doit traiter un parent comme l'un de ses enfants, on a préfère avoir recours à `dynamic_cast`.

⇒ Le rôle de l'opérateur `dynamic_cast` est semblable à celui de `static_cast`, à la différence près que le `dynamic_cast` ne sert qu'à la conversion d'***indirections*** d'une classe à une autre. Les deux classes devant être apparentées, directement ou indirectement, et la validité de la conversion est vérifiée à l'exécution.

⇒ Remarque importante : `dynamic_cast` ne s'applique que quand le type original (avant transtypage) est polymorphique, et repose sur un ajout d'information dans les programmes (assurez-vous d'utiliser les options de compilation qui incluront `RTTI`, pour *Run-Time Type Information*, pour y avoir accès[68]). L'information pour réaliser la conversion est conservée dans la `vtbl` des objets.

Dans le cas où une conversion illégale est tentée, un `dynamic_cast` retournera `nullptr` pour une conversion illégale de pointeurs, et lèvera une exception (`std::bad_cast`, type tiré de `<typeinfo.h>`). Cette illégalité lors d'une tentative de conversion entre types non apparentés ou indirectement apparentés[69] est évaluée à l'exécution seulement.

Notez que `dynamic_cast` respecte les qualifications d'accessibilité. Par exemple, si une classe a un parent par héritage protégé, un `dynamic_cast` révélant cet état de fait succèdera pour un enfant mais pas pour une fonction globale (autre que `friend`).

---

[68] Ceci accroît la taille des programmes. En C++, on ne paie que pour ce qu'on utilise, alors par défaut cette option est à *Off* dans les compilateurs.

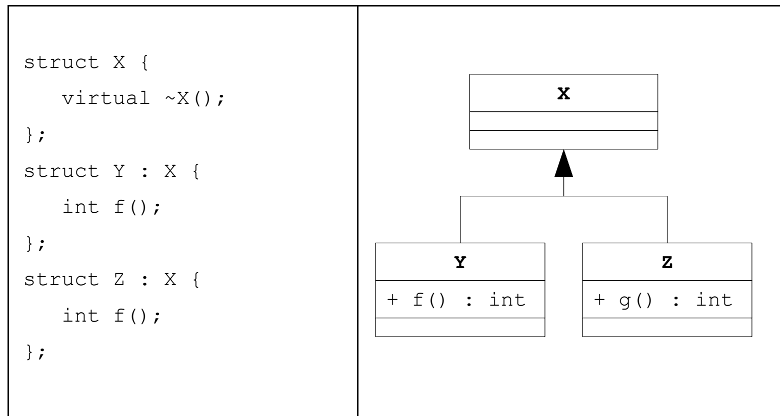
[69] Les *sibling classes*, ou classes partageant un même parent (comme le sont les classes `Y` et `Z` dans l'exemple illustrant le transtypage par `static_cast`). La légalité d'une conversion dans un tel cas variera vraiment en fonction de la dynamique du processus exécutant.

Validité dynamique et validation dynamique

L'opérateur de transtypage `dynamic_cast` réalise les opérations dont la légalité effective n'est connue qu'au moment de l'exécution des programmes. Conséquemment, en pratique, on vérifiera le résultat de cet opérateur avant de s'en servir.

À titre d'illustration, examinons la hiérarchie suivante, composée de trois classes : un parent (`X`) et deux enfants publics directs (`Y` et `Z`).

Nous avons enrichi les classes de l'exemple précédent; entre autres, la classe `X` est polymorphique, dû à l'ajout d'un destructeur virtuel, ce qui implique que ses enfants `Y` et `Z` le sont aussi.



```
struct X {
   virtual ~X();
};
struct Y : X {
   int f();
};
struct Z : X {
   int f();
};
```


L'exemple ci-dessous se base sur cette hiérarchie.

À droite, `main()` cherche à utiliser un `Y` à travers un pointeur de `Z`. Un `static_cast` interdirait cette tentative de transtypage, faute de lien direct entre les classes. Et pour cause : `Y` expose `f()` mais pas `g()`; traiter un `Y` comme un `Z` est donc dangereux.

Ici, puisque la conversion est faite sur des pointeurs, une tentative de conversion illégale mènera à un pointeur nul.

Par la suite, une tentative est faite de considérer `y` (de type `Y`) comme une référence sur un `Z`. Ici, puisque la conversion est faite sur des références, une tentative de conversion illégale résultera en la levée d'une exception de type `std::bad_cast`.

Évidemment, dans un cas comme dans l'autre, si la conversion est légale, alors l'indirection obtenue est utilisable et le code s'exécute normalement.

```
#include <typeinfo>
int main() {
   using std::bad_cast;
   Y y;
   Z *pz = dynamic_cast<Z*>(&y);
   if (pz) // sera faux
      z-> g();


   try {
      Z &rz = dynamic_cast<Z&>(y);
      z.g();
   } catch (bad_cast&) {
   }
}
```

Notez qu'il est possible de restreindre la portée du pointeur obtenu par un `dynamic_cast` à l'alternative qui s'en servira en le déclarant à même l'alternative :

```
if (auto pz = dynamic_cast<Z*>(&y), pz) { // obtenir pz, puis le tester
   // ... la portée de pz est celle du « if »
}
```

Ceci évite parfois de laisser traîner des variables dont la durée de vie utile est limitée à une zone clairement définie du programme.

En résumé

Une opération de transtypage est un mensonge, qui sert à transformer le regard que pose le compilateur sur certaines entités d'un programme. Ce changement de regard ne transforme pas les objets, mais peut créer des objets temporaires et, règle générale, modifie temporairement (pour la durée d'une expression) la perception que le programme a des types impliqués.

Les opérateurs de transtypage clés pour de C++ sont `static_cast` et `dynamic_cast`. Comme leur nom l'indique, l'un fait son travail à la compilation et l'autre agit à l'exécution. Sans surprises, quand le travail est fait à la compilation, l'exécution est beaucoup plus rapide. Sans surprises aussi, quand le travail est fait à l'exécution, la conversion peut exploiter de l'information qui n'est connue qu'à ce stade, et peut donc détecter des problèmes qui resteraient indétectables à la compilation.

On utilisera `static_cast` pour les conversions naturelles, implicites, ou pour les conversions d'une indirection vers un parent à une indirection vers un enfant. Toutefois, les conversions de parent à enfant sont risquées, et `static_cast` ne devrait être utilisé que dans les cas où les programmeuses et les programmeurs sont certain(e)s qu'il s'agit d'une opération sans danger.

On utilisera `dynamic_cast` pour les conversions entre classes apparentées de près ou de loin, qu'il s'agisse de convertir d'une classe à l'autre quand les deux classes partagent un parent ou quand on souhaite passer d'un parent à un enfant sans avoir la certitude que la conversion est saine. Cet opérateur ne s'applique qu'à des types polymorphiques, et y avoir recours implique accroître la taille des programmes compilés. L'inférence dynamique de types que réalise cet opérateur est une opération de navigation de graphe, et est nettement plus lente qu'une conversion réalisée à la compilation.

Par contre, cette conversion est plus sécuritaire que `static_cast` car elle détecte les erreurs de types de manière vérifiable par programmation (en testant un pointeur ou en réalisant de la gestion d'exceptions).

En bref

Un `dynamic_cast` intervient lorsque le programme tente d'utiliser un objet polymorphique d'une manière brisant l'encapsulation; en théorie, cela ne devrait jamais se produire, mais en pratique, de tels cas existent.

Dans d'autres langages

Les langages comme Java, C# et VB.NET qui ne supportent que l'héritage simple ont plus besoin des conversions explicites de types que les langages supportant l'héritage multiple comme C++, CLOS, Perl et Eiffel.

En Java, la syntaxe des conversions explicites de types suit celle du langage C, c'est-à-dire l'insertion devant l'objet à convertir du type de destination entouré par une paire de parenthèses. La même syntaxe permet la conversion explicite de données dont le type est primitif.

La conversion d'un dérivé en l'une de ses bases est sans danger. Une conversion d'une base en l'un de ses dérivés possibles peut lever un ClassCastException. Une conversion en un type non apparenté sera détectée comme illégale dès la compilation.

```
class Base {
}
class Dérivé extends Base {
}
class Étrange {
}
public class Z {
   public static void main(String [] args) {
      Base b = new Dérivé();
      Dérivé d = (Dérivé) b;
      b = d;
      b = (Base) d;
      Étrange e = (Étrange) b; // illégal à la compilation
      b = new Base();
      // illégal à l'exécution (lève un ClassCastException)
      d = (Dérivé) b;
   }
}
```

En C#, la situation est exactement la même, de la syntaxe des conversions aux règles qui l’accompagnent, pour les objets comme pour les types primitifs.

La seule légère nuance entre java et C# de ce côté tient au nom de l’exception levée lors d’une tentative de conversion entre classes non apparentées (celle de C# se nomme `InvalidCastException`).

```
namespace z
{
   class Base
   {
   }
   class Dérivé : Base
   {
   }
   class Étrange
   {
   }
   public class Z
   {
      public static void Main(string [] args)
      {
         Base b = new Dérivé();
         Dérivé d = (Dérivé) b;
         b = d;
         b = (Base) d;
         // illégal à la compilation
         Étrange e = (Étrange) b;
         b = new Base();
         // illégal, à l'exécution (InvalidCastException)
         d = (Dérivé) b;
      }
   }
}
```

En VB.NET, il existe une fonction primitive (connue par le langage) pour convertir tout type primitif en un autre type primitif (CInt(x), par exemple, convertit x en Integer s'il est possible de le faire).

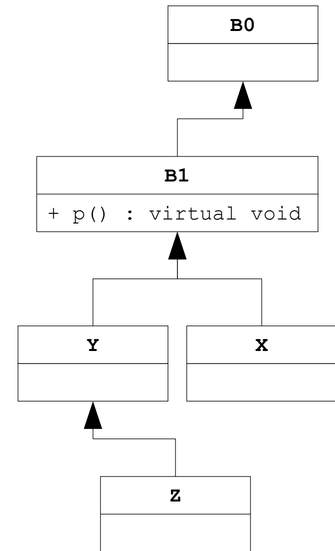
Considérant qu'il soit impossible de couvrir de manière semblable tous les cas de conversions possibles pour les classes susceptibles d'exister un jour, la conversion d'un objet d'une classe à l'autre se fait en invoquant la fonction primitive à deux opérandes nommée **CType()**.

```
Namespace z
   Public Class Base
   End Class
   Public Class Dérivé
      Inherits Base
   End Class
   Public Class Étrange
   End Class
   Public Class Z
      Public Shared Sub main()
         Dim b As Base = New Dérivé
         Dim d As Dérivé = CType(b, Dérivé)
         b = d
         b = CType(d, Base)
         ' illégal dès la compilation
         Dim e As Étrange = CType(b, Étrange)
         b = New Base
         ' illégal à l'exécution (lève un InvalidCastException)
         d = CType(b, Dérivé)
      End Sub
   End Class
End Namespace
```

La fonction CType() cherche à convertir le premier paramètre dans le type représenté par le deuxième paramètre. Étant une fonction primitive et prise en charge par le compilateur VB.NET lui-même, elle est en mesure de détecter les tentatives de conversions entre types non apparentés dès la compilation.

Exercices – Série 05

Étant donné la hiérarchie visible à droite, où l'héritage est dans chaque cas public, indiquez pour chacune des conversions ci-dessous :



- si on peut procéder à la conversion implicitement (sans appliquer d'opérateur de conversion explicite de types);
- si on peut utiliser un `static_cast`; et
- si on devrait utiliser un `static_cast`;
- si on peut utiliser un `dynamic_cast`; et
- si on devrait utiliser un `dynamic_cast` (ceci inclut les cas où la conversion est clairement dangereuse).

Il est possible que plusieurs réponses s'appliquent pour une conversion donnée.

> Il y a une nuance entre *pouvoir* et *devoir* dans l'application d'opérateurs de conversion explicite de types.

Conversion tentée	Implicitement	`static_cast` possible	`static_cast` préférable	`dynamic_cast` possible	`dynamic_cast` préférable
Convertir un `X*` en `Y*`	☐	☐	☐	☐	☐
Convertir un `Z*` en `X*`	☐	☐	☐	☐	☐
Convertir un `X*` en `Z*`	☐	☐	☐	☐	☐
Convertir un `Y*` en `B1*`	☐	☐	☐	☐	☐
Convertir un `B1*` en `Y*`	☐	☐	☐	☐	☐
Convertir un `B1*` en `B0*`	☐	☐	☐	☐	☐
Convertir un `B0*` en `B1*`	☐	☐	☐	☐	☐
Convertir un `Z*` en `B0*`	☐	☐	☐	☐	☐

Autres opérateurs de transtypage `ISO`

Nous examinerons rapidement ici les deux autres opérateurs de conversion explicite de types `ISO` que sont le `reinterpret_cast` et le `const_cast`. Bien qu'ils apparaissent moins souvent dans les programmes `OO` que les opérateurs `static_cast` et `dynamic_cast` vus précédemment, ces opérateurs jouent un rôle précis et parfois essentiel dans la rédaction de programmes sérieux.

Les rôles que jouent ces opérateurs méritent réflexion, même si vous utilisez des langages qui offrent moins de latitude de pensée aux programmeuses et aux programmeurs. Les comprendre amène à mieux comprendre les opérations de transtypage dans l'ensemble, et soulève des questions quant aux raisons qui motivent ces opérations.

Fermer les yeux : `reinterpret_cast`

Il arrive qu'un programme doive mentir sur la nature de quelque chose de brut, sans le moindre soutien sémantique. Par exemple, traiter un tableau de `int` comme un tableau de `short`, ou traiter un pointeur sur une instance de `X` comme un pointeur sur une instance de `Y` sans que les ceux classes n'aient le moindre lien entre elles. De tels mensonges tombent sous l'égide de l'opérateur de transtypage `reinterpret_cast`.

⇒ L'opérateur `reinterpret_cast` indique au compilateur de se fermer les yeux et de traiter une indirection vers un type comme une indirection vers un type qui n'y est absolument pas apparenté. Il ne s'applique qu'aux indirections (pointeurs et références).

Ces mensonges apparaissent souvent en programmation système, où les programmes manipulent des adresses brutes (`void*`; nous y reviendrons sous peu). Bien que très brutes, très « bas niveau », ces conversions permettent à des programmes `OO` d'interagir avec des systèmes plus primitifs (des pilotes de périphériques, par exemple, ou certaines bibliothèques rédigées dans d'autres langages) et permet d'intégrer efficacement ces systèmes à un monde `OO`.

Il est peu probable qu'un(e) gestionnaire de projet ait recours à `reinterpret_cast`, mais comprendre le rôle de cet opérateur (et savoir qu'il existe) permet, de manière pragmatique, de faire le pont entre une technologie `OO` et des technologies qui n'ont aucun lien avec elle.

On n'utilisera l'opérateur `reinterpret_cast` que si nous sommes dans nos derniers retranchements. Il s'agit d'une opération qu'on doit chercher à éradiquer de tout programme, ou du moins à encadrer de manière très stricte, et qui doit entre-temps être très documentée.

Cet opérateur permet des conversions dénaturées, *contre nature*. Toute manipulation de l'objet pointé résultant d'un `reinterpret_cast` est une opération à haut risque et réduit la portabilité des programmes.

> Dans l'exemple ci-dessous, nous traiterons un `short` comme s'il s'agissait de deux `bool` consécutifs en mémoire, donc d'un tableau de `bool`, et nous présumerons d'un ordre des *bytes* en mémoire dans le `short`. Cet exemple est dangereux et non portable pour plusieurs raisons; ne faites pas ça à la maison. C'est d'ailleurs souvent le cas avec des manipulations d'aussi bas niveau, ce qui explique à la fois qu'elles soient permises et qu'on les évite à moins de vraiment savoir ce qu'on fait. Si l'exemple est trop technique à vos yeux, escamotez-le, tout simplement.

Un exemple d'utilisation est un objet qui doit manipuler une structure primitive, comme un pilote pour un périphérique. Dans notre situation, simplifiée à l'extrême, une entité matérielle (un `Cossin`, disons, pour lui donner un nom scientifique) sera représentée en mémoire par deux *bytes* contigus.

Imaginons que les deux *bytes* soient en fait des booléens (on peut écrire un `1`, ou `true`, sur le `1`er *byte* pour changer l'état d'une petite lumière et on peut consulter le `2`e *byte* pour savoir si la lumière est allumée ou non).

La classe `Cossin`, à droite, pur type valeur, encapsule cette structure primitive et expose des méthodes pour encapsuler la consultation et modifier l'état de la lumière.

Notez que `Cossin` pose des problèmes de portabilité, l'ordre des *bytes* dans un entier variant selon les plateformes. Les résultats non-portables sont d'ailleurs fréquents avec `reinterpret_cast`.

```
class Cossin {
   short &etat_; // présume sizeof(short)==2
public:
   Cossin(short &src) noexcept : etat_{src} {
   }
   bool est_allume()const noexcept {
      return reinterpret_cast<bool*>(&etat_)[1];
   }
   bool est_eteint() const noexcept {
      return !est_allume();
   }
   void basculer() noexcept {
      reinterpret_cast<bool*>(&etat_)[0] = true;
   }
}
```

Une instance de cette classe se saisit à la construction d'une référence sur un `short` et dissimule, de manière encapsulée, les opérations primitives dans des méthodes. Ceci permet d'exprimer naturellement les opérations pertinentes, et de localiser les ajustements requis lors de migrations d'une plateforme à l'autre.

Le type `void*`

Depuis les beaux jours du langage C[70], on utilise le type `void*` pour dénoter un pointeur vraiment abstrait : un pointeur duquel on ne sait rien d'autre que le fait qu'il représente une adresse en mémoire. Un pur lieu, vide de toute sémantique particulière. Ce niveau d'abstraction est inaccessible (par choix) en Java, par exemple, car il permet d'échapper aux règles sémantiques du système de types et donne un peu trop de pouvoir aux programmeuses et aux programmeurs, du moins selon la philosophie de ce langage.

Le type `void*` sert souvent pour représenter un paramètre dans un sous-programme s'appliquant à toute adresse en mémoire, peu importe son type. Certaines fonctions des bibliothèques standard du langage C, comme `std::memcpy()` (qui copie une plage mémoire sur une autre), `std::memset()` (qui remplit une zone mémoire d'une certaine taille avec une valeur entière donnée) ou `std::memcmp()` (qui compare, *byte* pour *byte*, deux zones mémoire) utilisent ce type pour des fins d'abstraction.

Les fonctions de base des principaux systèmes d'exploitation sont en général friandes de ce genre d'abstraction pure et dure. Étant donné son importance primordiale et son omniprésence, et considérant que C++ se fait un devoir de permettre l'écriture de programmes aussi efficaces que possible, on comprendra que ce type fait partie des mœurs.

En langage C, tout pointeur peut *implicitement* être considéré comme une adresse pure (un void*). En C++, tout pointeur peut être considéré comme un `void*` *à travers un transtypage par* `static_cast`. C'est une des différences fondamentales entre ces deux langages.

Un pointeur est une adresse typée, mais `void*` est une adresse pure. Manipuler des adresses pures signifie s'affranchir du système de types : seuls les programmeuses et les programmeurs peuvent savoir ce qu'il y a à une adresse donnée si la sémantique a été évacuée des adresses. Conséquemment, `void*` est à la fois l'entité la plus abstraite qui soit en C++ (loin au-delà des types comme `Object` en Java ou dans le monde `.NET`, qui sont des types sémantiquement riches), et l'entité la plus dangereuse.

Un `void*` ne peut être utilisé pour lui-même, n'ayant d'autre sens que celui de lieu, en C comme en C++. Ce type doit donc être converti dans un type muni de sémantique avant qu'un programme ne puisse s'en servir.

En bref
Un `reinterpret_cast` signifie seulement « change ton point de vue sur le type à cette adresse ». Utilisez-le avec prudence et parcimonie.

[70] Mais pas depuis la première version de C, par contre. Au tout début, le type le plus abstrait de ce langage était `char*`, ou pointeur sur un *byte*.

Tricher sur la protection : `const_cast`

L'opérateur `const_cast` sert à retirer *temporairement*, soit pour la durée de l'opération de conversion, et *visiblement*, l'opérateur étant facilement repérable et identifiable, une qualification `const`[71]. C'est là un rôle très spécialisé, que même `reinterpret_cast` ne peut jouer, mais qui a sa raison d'être.

⇒ Pour procéder à un `const_cast`, il faut que le type original et le type dans lequel convertir soient *précisément* les mêmes, aux qualifications `const` et `volatile` près.

Comme les autres opérateurs de transtypage (outre `dynamic_cast`, bien entendu), l'opérateur `const_cast` s'applique à la compilation et n'a que peu ou pas de coût à l'exécution.

```
class Rectangle {
   // ...
public:
   // pas const; flute!
   void dessiner();
   // ...
};
```

Illustration : soit une constante de type `Rectangle`, qu'on désire dessiner. On sait que la méthode `dessiner()` du `Rectangle` ne modifie pas le rectangle qu'elle dessine, mais (pour une raison qui nous échappe) les développeurs de la classe `Rectangle` n'ont pas cru bon de qualifier sa méthode `dessiner()` de `const`[72].

Il nous est donc impossible d'appeler la méthode `dessiner()` de notre `Rectangle` constant, ce qui est fort ennuyeux. À long terme, on souhaitera que la faute dans la classe `Rectangle`, qui fut de négliger l'application de la spécification `const` à une méthode qui aurait dû l'être, soit réparée.

```
void f() {
   const Rectangle r(10, 20);
   // serait légal seulement si
   // dessiner() était const
   r.dessiner(); // ne compile pas
}
```

Entre-temps, on pourra quand même dessiner un `Rectangle` en appliquant un `const_cast` sur notre constante le temps de l'appel.

Dans cet exemple, le `Rectangle` nommé `r` est considéré non constant seulement pour la durée de l'appel à `dessiner()`. Il faut agir de manière responsable pendant ce moment critique (éviter un appel à un mutateur, par exemple).

```
// version opérationnelle...
void f() {
   const Rectangle r{10, 20};
   const_cast<Rectangle&>(r).dessiner();
}
```

Remarquez le recours à une référence dans la notation du type de destination. Un `const_cast` n'a de sens que si l'objet visé est converti en indirection non constante; autrement, créer une variable temporaire qui copierait l'original suffirait.

[71] … ou la qualification `volatile`, que nous n'examinerons pas ici. Le sujet est passionnant mais repose sur des considérations de multiprogrammation.

[72] Peut-être ne savaient-ils pas qu'il est possible de spécifier une méthode `const`, qui sait? Faudrait les inviter à suivre un de ces excellents cours de `POO` offerts par nos chics institutions d'enseignement québécoises, ne pensez-vous pas?

Le risque

On constate que le `const_cast` permet entre autres de contourner certaines fautes de design, certains oublis quant aux qualifications `const` ou `volatile`. Il permet aussi de réaliser des interfaces entre technologies supportant les objets constants (C++) et d'autres dont le système de types n'offre pas ce raffinement (Java et les langages `.NET`, entre autres).

Évidemment, il y a un risque, car s'il s'avérait que la méthode appelée ne soit effectivement pas garante de la constance de l'objet (si elle a une chance de modifier l'objet), alors le fait que nous ayons forcé une *perte de constance* temporaire de cet objet peut avoir un impact dramatique.

La qualification `mutable`

Une des raisons pour lesquelles les gens ont parfois recours à un `const_cast` est pour oublier qu'un attribut est constant, même dans une méthode `const`.

Par exemple, si une classe `Rectangle` expose le service `dessiner()` qualifié `const`, et si la classe souhaite comptabiliser le nombre d'invocations à cette méthode pour un objet donné, elle frappera un irritant : dans une méthode `const`, les attributs d'instance sont aussi `const` et on ne peut pas les modifier.

```
class Rectangle {
   int nappels_ = {};
   // ...
public:
   Rectangle()= default;
   // ...
   void dessiner() const {
      ++nappels_; // illégal!
      // ...
   }
};
```

N'oubliez pas que `const`, dans la signature d'une méthode, s'applique à `this`, donc l'expression `++nappels_` équivaut à `++(this->nappels_)` ce qui, visiblement, constitue un bris de constance.

Appliquer un `const_cast` à `nappels_` dans `dessiner()` est une possibilité, mais communique mal l'intention; nous éviterons donc cette option. Retirer la qualification `const` de la méthode réduit son applicabilité, et n'est pas non plus une option.

Heureusement, il existe une alternative. Lorsqu'un attribut d'instance est tel qu'il doit en tout temps échapper à la qualification `const`, typiquement parce que cet attribut joue un rôle interne et qui n'influence pas l'interface publique de l'objet (ici, comptabiliser les appels est un détail d'implémentation, pas une caractéristique fondamentale d'un `Rectangle`), le langage C++ offre le mot clé ***`mutable`***.

⇒ Un attribut d'instance **`mutable`** est un attribut qui ne sera jamais soumis à la spécification `const`, et ce peu importe le contexte ou la situation dans laquelle est utilisée l'instance qui en est propriétaire.

Le mot `mutable` est une reconnaissance conceptuelle de la différence qu'il y a entre faire en sorte que les états pertinents d'un objet soient constants pour une durée donnée (vie de l'objet, invocation d'une méthode, invocation du sous-programme utilisant l'objet, *etc.*) et certains objectifs pragmatiques qui font partie de ce qu'encapsule l'objet mais ne font pas partie de ce qu'il représente fondamentalement. La différence entre le concept représenté et certains trucs techniques internes permettant de bien le faire.

Notre exemple est comptable, permettant à un objet de tenir à jour de l'information destinée strictement à des fins d'usage interne pour comptabiliser des statistiques d'utilisation, mais un autre cas typique de recours à mutable est de permettre l'implantation d'une forme d'antémémoire (mémoire *cache*) dans le but d'optimiser certains appels ultérieurs.

Identifier une donnée comme `mutable` permet d'écrire du code à la fois robuste (respect plein et entier de l'encapsulation[73]) et pragmatique (flexibilité à l'interne quant à l'application des règles et contraintes de constance).

Une nouvelle version de notre exemple, adaptée à l'aide d'une qualification `mutable` sur l'attribut `nappels_`, est proposée à droite. Appliquer cette qualification à l'attribut fait en sorte que, dans `dessiner()`, `this` soit `const` mais que l'attribut `this->nappels_`, lui, ne le soit pas.

```
class Rectangle {
   mutable int nappels_ {};
   // ...
public:
   Rectangle()= default;
   // ...
   void dessiner() const {
      ++nappels_; // Ok
      // ...
   }
};
```

On comprendra qu'il faille éviter d'abuser du mot clé `mutable`, ce qui deviendrait vite une nuisance au principe d'encapsulation.

Dans une méthode `const` comme dans un objet constant, on souhaite typiquement le respect de la constance; donc que les attributs d'instance soient constants; les cas qui échappent à ce souhait sont peu nombreux et leur raison doit être documentée.

[73] Cette affirmation a fait réagir certains de mes estimés collègues et amis, et je comprends leur réaction. L'idée que je veux mettre de l'avant ici est que l'objet veut protéger ses états (on me souligne que états *fondamentaux* aurait peut-être été moins sujet à polémique ici), mais que certains attributs d'instance, bien qu'ils soient utiles à l'objet, ne sont pas des états au sens où on l'entend.

Dans le cas du `Rectangle`, le décompte statistique des invocations de chaque catégorie de service est un état temporaire et qui n'est pas tant un état de `Rectangle` qu'un état utilitaire (et peut-être transitoire) pour la mise au point du programme. Dans les langages qui le permettent, de la programmation orientée aspect, ou `POA` [POOv03], aurait d'ailleurs pu être appliquée.

Conceptuellement, les véritables états d'un objet devraient rester constants lors de l'invocation d'une méthode constante (il y a peut-être des cas d'exception, mais ceux-ci devraient être rares et pointus); on s'attend à ce que les attributs mutables soient auxiliaires et disjoints de la nature de l'objet représenté. C'est ce qui explique mon choix de parler de *respect plein et entier de l'encapsulation* ici : au sens où je l'entends dans ce passage, un attribut comme `nappels_` n'est pas un état réel de `Rectangle`. Cela dit, je comprends le caractère polémique de l'affirmation et je ne prétends pas avoir trouvé la meilleure formule pour exprimer l'idée que je cherche à mettre de l'avant dans ce passage, alors les suggestions sont les bienvenues.

En résumé

En plus des opérateurs de transtypage `static_cast` et `dynamic_cast`, C++ ISO offre aussi l'opérateur `reinterpret_cast`, qui permet de tromper le compilateur quant au type d'une indirection, et l'opérateur `const_cast`, qui permet d'ajouter ou d'oublier momentanément les qualifications `const` ou `volatile`.

Ces deux opérateurs jouent des rôles pointus dans l'écosystème du langage. Le premier permet, entre autres, de travailler à très bas niveau, donc d'intégrer des technologies disparates, mais demande des aptitudes techniques pointues puisque jouer à un niveau aussi bas tend à produire du code non portable. Le second permet entre autres d'adapter des technologies qui ont des systèmes de types moins complets à un programme OO plus strict.

L'abstraction la plus haute de C++ est l'adresse pure, ou `void*`. Traiter un pointeur comme une adresse pure (ou inversement) implique un `static_cast`. Une adresse pure est exempte de toute sémantique de type, échappant donc au filet de sécurité du compilateur, et ne représente qu'un lieu en mémoire; manipuler des adresses brutes est dangereux et puissant, il ne faut donc pas en abuser.

Lorsqu'un attribut d'instance doit échapper à une qualification de constance, on peut le qualifier de `mutable`.

En bref

Un `const_cast` signifie que les qualifications de sécurité en vigueur peuvent être changées sans risque. Le concept-même est préoccupant, mais l'opération est pragmatique. Visez à accompagner ces opérations de commentaires.

Dans d'autres langages

Nous ne reviendrons pas ici sur les conversions explicites de types de Java et des langages `.NET`, le tout ayant été couvert précédemment. Notons peut-être brièvement que C# permet de manipuler des pointeurs un peu comme le langage C, mais dans des blocs d'instructions explicitement qualifiés `unsafe`. Les assemblages contenant des blocs `unsafe` peuvent être rejetés par certains systèmes.

Le concept de mutabilité n'a de contrepartie en Java ou dans les langages `NET`, aucun d'eux ne supportant à ce jour les objets constants.

Tout langage proposant des mécanismes pour hiérarchiser des classes se livre à un questionnement philosophique : sachant que le parent est une abstraction de plus haut niveau que l'enfant, y a-t-il une abstraction plus haute que toutes les autres et, si oui, quelle est-elle?

Sous Java, la réponse est oui, et cette abstraction est la classe `Object`. Cette classe représente le concept d'objet (pris au sens *d'instance de quelque chose*), et Java a fait le choix de faire de la classe `Object` une abstraction plus fondamentale que ne l'est la classe `Class`.

La classe `Class` de Java est ce qu'on nomme une ***métaclasse*** [POOv03].

En effet, la bibliothèque de classes standards de Java propose une classe `Class` qui dérive d'`Object` et dont les instances sont des classes.

Dans les langages `.NET`, la réponse est aussi oui et l'abstraction s'y nomme aussi `Object`. La classe `Object` joue pour les langages `.NET` le même rôle qu'en Java, mais avec des différences fines (qui méritent une lecture approfondie de la documentation de chacune de ces deux plateformes). Le type `Object` de la plateforme `.NET` est plus abstrait que le type `Type`, qui est l'équivalent de la classe `Class` de Java.

Héritage multiple

Pourquoi cette section?

L'héritage multiple est, encore aujourd'hui, controversé. Pourtant, il s'agit d'une idée ayant servi de racine à des outils parmi les plus prisés des informaticien(ne)s contemporain(e)s, et qui permet de réaliser l'approche maintenant répandue d'héritage d'interfaces. Son *bon* usage est une considération de design fondamentale. Pour un langage OO, le choix de supporter l'héritage simple d'implémentation et l'héritage multiple d'interfaces (Java, C#) ou de supporter pleinement l'héritage multiple à la fois d'implémentation et d'interfaces (C++, Eiffel, Python, CLOS) influence grandement l'ensemble des options de développement et de pensée disponibles.

Dans cette section, nous examinerons comment se gère l'***héritage multiple*** en C++, incluant les très importantes (et très répandues) ***interfaces***, ce mot étant pris ici dans un sens technique. Même les langages qui ne supportent pas l'héritage multiple (dont Java et les langages .NET) supportent au moins l'héritage d'interfaces.

Le langage C++, comme à peu près *tous* les langages se présentant comme OO, supporte le principe d'héritage *au sens simple* : permettre à une classe d'avoir un parent dont elle hérite l'accès direct aux membres publics et protégés.

Le langage C++, comme *certains* langages OO, supporte aussi l'héritage *multiple*, soit la possibilité pour une classe d'avoir plusieurs parents. Ceci peut être un atout de design important, mais peut aussi être nuisible, tout dépendant de l'approche avec laquelle on l'appliquera.

On examinera donc, dans cette section, comment utiliser l'héritage multiple, mais surtout comment l'utiliser avec sagesse, et comment en profiter.

Arguments contre l'héritage multiple[74]

Il y a une vogue de longue date selon laquelle l'héritage multiple est une tare, vogue à laquelle j'ai moi-même souscrit assez longtemps. La position générale de celles et ceux s'opposant à cette idée est liée à un ou plusieurs des arguments suivants :

- l'héritage multiple ralentit les programmes;
- l'héritage multiple est *superflu* – l'héritage simple suffit;
- l'héritage multiple *grossit les programmes*; ou encore
- l'héritage multiple complique les programmes.

Examinons ces arguments un à un.

[74] Pour d'autres arguments et d'autres exemples d'utilisation d'héritage multiple avec C++, voir [HMult]. De même, pour un commentaire sur le caractère dommageable de restreindre l'héritage multiple aux interfaces seules dans certains langages, voir [ItfHarm].

De la relation entre héritage multiple et vitesse

Le premier point est facile à démentir, du moins en C++ : accéder aux membres d'un parent unique ou à ceux d'un parent parmi plusieurs implique les mêmes coûts en terme de vitesse.

L'argument est fallacieux, à part peut-être sur des compilateurs particuliers (C++ ou autres) pour lesquels l'implantation de l'héritage multiple est déficiente. Il n'y a pas, en C++, de coûts de performance particuliers à l'exécution de programmes employant l'héritage multiple, du moins en général.

> Qu'une classe `D` ait pour seul parent `B` ou qu'elle ait deux parents `B0` et `B1`, l'effort requis pour que le programme réfère aux membres de `B` dans un cas ou à ceux de `B0` ou de `B1` dans l'autre cas est le même.

Il peut y avoir des coûts particuliers dans quelques cas :

- tel que mentionné plus haut, certains compilateurs implémentent moins efficacement l'héritage multiple qu'ils ne le pourraient. Une raison pour cette différence est que l'héritage multiple est moins connu, donc moins répandu, que son cousin;
- certains langages `OO` dynamiques, en particulier ceux qui permettent de modifier dynamiquement les relations d'héritage (à titre d'exemple, certaines techniques JavaScript peuvent permettre d'agir ainsi), peuvent effectivement avoir des implémentations plus coûteuses à l'exécution pour l'héritage multiple que pour l'héritage simple;
- plusieurs compilateurs peinent à appliquer `EBCO` (voir ***Héritage et consommation d'espace***) dans des situations d'héritage multiple (mais il existe une solution à ce problème[75]).

Aujourd'hui, les compilateurs C++, pour la majorité, gèrent correctement l'héritage multiple. Cette approche est utilisée même dans la bibliothèque standard du langage, qui ne fait aucun compromis sur la vitesse d'exécution, ce qui peut être considéré comme un *imprimatur*.

Du caractère superflu de l'héritage multiple

Le deuxième point est délicat. En effet, on peut voir que l'héritage simple (augmenté de l'héritage d'interfaces) peut suffire à combler l'essentiel des besoins de développement `OO` en regardant le nombre de langages `OO` qui s'y limitent.

Par contre, poussant le raisonnement plus loin, on peut aussi constater qu'on arrivait en général à faire ce qu'on voulait sans héritage du tout, dans le modèle structuré. Cela ne nous empêche pas d'apprécier les atouts propres au modèle objet, et à implanter l'héritage simple. Pourquoi en serait-il autrement pour l'héritage multiple? Sans qu'il s'agisse d'une panacée, il faut convenir qu'il s'agit d'un outil de plus dans le coffre à outils des programmeuses et des programmeurs, et réfléchir non pas à la pertinence ou non de l'outil, mais bien aux manières par lesquelles cet outil peut améliorer l'expérience de programmation et notre pensée `OO`.

---

[75] Voir [hdEnPar] pour les techniques impliquées et des exemples.

De la relation entre héritage multiple et espace

Le troisième point *peut* s'avérer vrai, entre autres en ce sens général qu'une application imprudente d'héritage, même *simple*, peut faire grossir un programme. Tout héritage implique un coût en espace, croissance linéaire en fonction de la taille des parents; le même coût, en fait, que dans une conception par composition.

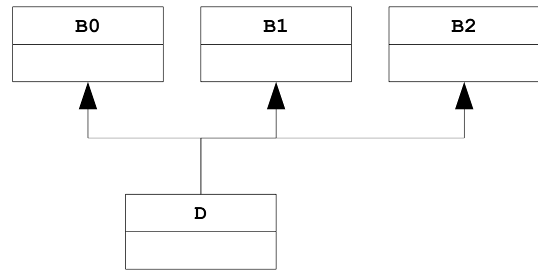
Il y a des bémols au passage : l'utilisation ou non d'héritage *virtuel* (voir plus loin) influence le rapport entre la taille d'un objet et le nombre de ses parents, mais d'une manière dépendante du design choisi. Tel que mentionné plus haut, `EBCO` tend à être moins bien réussie par les compilateurs en situation d'héritage multiple qu'en situation d'héritage simple, mais il s'agit d'une difficulté technique, pas d'une faute conceptuelle

De la relation entre héritage multiple et complexité

Le quatrième peut *aussi* être vrai, mais pour toute technologie, toute approche, tout outil. L'héritage multiple est une technique (parmi plusieurs) avec laquelle on peut grandement compliquer un programme si on est imprudent(e). Heureusement, la complexité potentielle tient à l'usage fait de la technique, pas de la technique en soi.

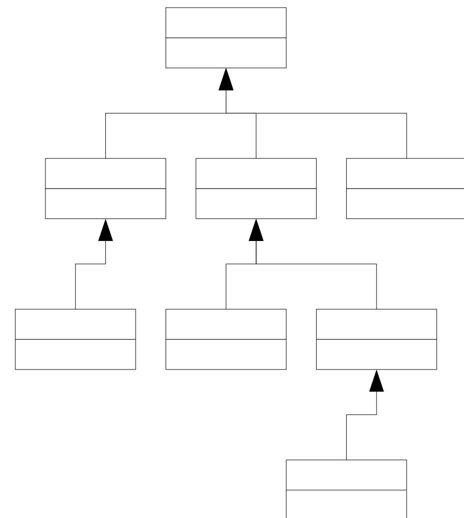
L'héritage multiple peut aussi nous aider à simplifier des hiérarchies de classes, et surtout à les aplanir; nous le verrons dans [POOv03] lorsque nous discuterons de l'équivalence entre concepts et techniques dans le monde `OO` : si une opportunité intéressante de design reposant sur l'héritage multiple se fait sentir et si le concept n'existe pas dans le langage `OO` choisi, alors les manœuvres requises pour réaliser le design seront bien plus compliquées que ne le serait celles qui auraient servi à mettre au point la solution équivalente par héritage multiple.

L'idée



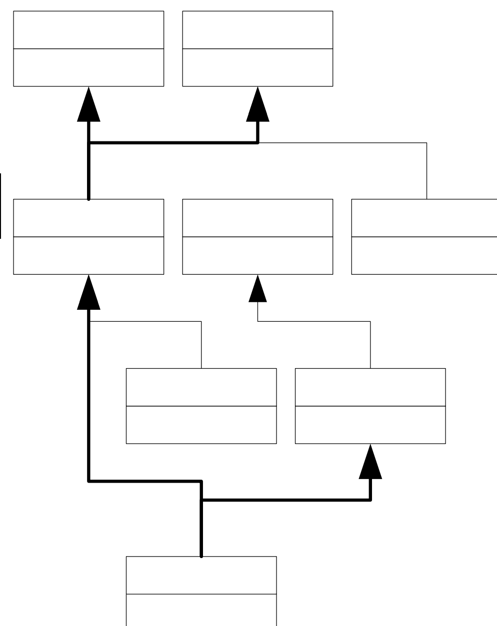
L'idée derrière l'héritage multiple est de permettre à une classe d'avoir plus d'un parent. Ainsi, si une classe `D` dérive des classes `B0`, `B1` et `B2`, alors :

- toute instance de `D` est tout ce qu'une instance de `B0` est, en plus de ce qui la rend spéciale par rapport à un `B0`;
- toute instance de `D` est tout ce qu'une instance de `B1` est, en plus de ce qui distingue d'un `B1`; et
- toute instance de `D` est tout ce qu'une instance de `B2` est, en plus (évidemment) de ce qui la rend spéciale en comparaison avec un `B2`.



Les règles habituelles s'appliquent : l'enfant gagne l'accès immédiat à tous les membres publics et protégés de chacun de ses parents, et peut être traité, par le code client, comme un cas particulier de chacun de ses parents publics.

On remarquera (par exemple dans le schéma proposé à droite) que l'héritage simple, où chaque enfant n'a qu'un seul parent, crée une hiérarchie de classes en *arbre*.



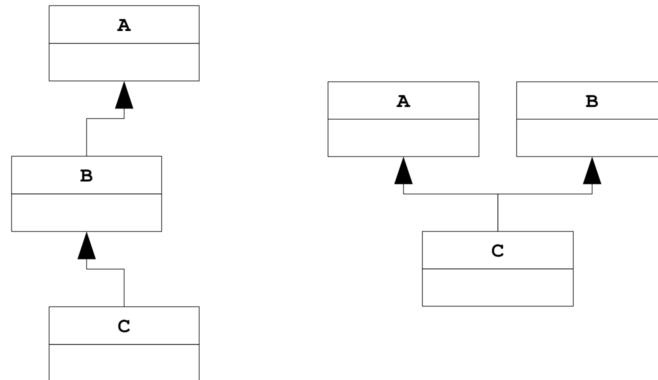
Pour sa part, l'héritage multiple, une fois introduit, entraîne plutôt une forme plus générale de *graphe* ou de treillis (en anglais : *Lattice*).

> Dans le schéma à droite, les classes dérivant par héritage multiple apparaissent comme descendant d'au moins deux traits gras.

Certaines structures ne posent pas vraiment de problèmes, alors que d'autres sont relativement subtiles (ce sont les structures complexes et subtiles qui angoissent les gens qui craignent l'héritage multiple).

Avoir la possibilité de construire des hiérarchies complexes en forme de treillis ne signifie pas avoir l'obligation de construire des structures aussi complexes. La plupart des structures conçues à partir d'héritage multiple sont plutôt simples et plates.

Examinons par exemple les deux hiérarchies ci-dessous. L'une (à gauche) repose strictement sur l'héritage simple et l'autre (à droite) repose sur l'héritage multiple.



On pourrait résumer, pour ces deux exemples, les similitudes et les différences entre ces deux représentations des classes A, B et C. Pour simplifier la discussion, nous présumerons que l'héritage est chaque fois public.

À gauche (héritage simple seulement)	À droite (héritage multiple)
Tout C est aussi à la fois un A et un B.	Tout C est aussi à la fois un A et un B.
Tout B est aussi un A.	A et B sont deux classes indépendantes l'une de l'autre.
Le choix de faire de tout C non seulement un B mais aussi un A est un choix fait par la classe B.	Le choix de faire de tout C non seulement un B mais aussi un A est un choix fait par la classe C.

Typiquement, dans un cas comme celui-ci, le choix entre héritage simple et héritage multiple se fait en fonction des besoins de la classe terminale (la classe la plus dérivée du lot, donc C). Si B n'a pas besoin de A, alors l'approche par héritage simple force une dépendance entre deux classes qui pourraient (ou devraient) ne pas en avoir. Évidemment, si un B est véritablement un A, alors l'héritage simple convient tout à fait et devrait être privilégié.

Notre exemple met en relief le fait que le couplage est plus fort dans un cas d'héritage simple que dans un cas d'héritage multiple (la dépendance entre A et B n'existe que dans l'approche par héritage simple seulement). Dans un cas comme dans l'autre, la classe C demeure telle quelle, mais le couplage entre les classes, au cumulatif, est plus faible avec l'héritage multiple.

Remarquez une analogie avec l'instanciation tardive : l'héritage multiple déplace le couplage aussi tard que possible et invite à concevoir des classes *a priori* indépendantes, associées dans un enfant à sa demande seulement, alors que l'héritage simple force la mise en place de dépendances entre classes selon des besoins présumés (A et B, règle générale, ne connaissent pas C).

Morale :	l'héritage simple est indiqué lorsque les relations hiérarchiques parent/ enfant sont nécessaires en soi. Si certaines relations sont forcées, il y a faute de design.

La syntaxe

On écrira, en C++, que la classe `D` hérite publiquement de `B0` ***et*** de `B1` comme ceci (à droite).

```
class D : public B0, public B1 {
   // détail de la classe D
};
```

On peut bien sûr hériter de manière *privée* ou *protégée* de l'un ou l'autre de nos parents, et mêler les trois spécifications de protection d'un parent à l'autre.

On ne peut pas hériter ***directement*** plus d'une fois du même parent. Le code à droite est donc **illégal**, puisque `D` dérive à la fois de `B` et de `B`. En effet, si une fonction `void f(B&);` recevait un `D` en paramètre, le code serait ambigu et il serait impossible de distinguer l'un des deux `B` dans ce `D` de l'autre.

```
class D : public B, public B {
   // ...
};
```

De même, si une instance de `D` voulait identifier son parent par son nom, invoquant par exemple `B::f()` pour une méthode `f()` de son parent `B`, l'expression demeurerait ambiguë.

Dans le code à droite, la classe `E` générera un avertissement à la compilation parce qu'elle dérive *directement* d'une classe qui est aussi un ancêtre d'un de ses parents. La raison est que tout accès à l'une ou l'autre des parties `B` d'un `E` exigerait alors d'être explicitement qualifiée comme telle (sinon elle serait ambiguë).

```
class B {
   // ...
};
class D : public B {
   // ...
};
class E : public B, public D {
   // ...
};
```

Par exemple, dans l'exemple ci-dessous (reposant sur les classes `B`, `D` et `E` à droite), les conversions explicites par l'opérateur de transtypage `static_cast` sont nécessaires pour lever les ambiguïtés. Il va sans dire que repenser le design serait préférable :

```
void f(B&);
int main() {
   E e;
   f(e); // ambigu
   f(static_cast<E::B&>(e));    // parent direct
   f(static_cast<E::D::B&>(e)); // parent indirect
   f(static_cast<E::D&>(e));    // idem
}
```

Par contre, ici, la classe `E` est tout à fait légale. Elle est dérivée directement de `D0` et de `D1`, tous deux dérivés de `B`, mais `E` elle-même ne dérive pas en même temps directement de `B` et d'un dérivé de `B`. De telles situations sont raisonnables et se produisent en pratique (nous en verrons des exemples plus loin).

Un tel design met en relief que les classes `D0` et `D1` ont toutes deux structurellement besoin de `B`, et qu'il est possible que les fonctionnalités de `D0` et de `D1` soient à la fois disjointes et utiles dans la conception de la classe `E`. Combiner `D0` et `D1` dans `E` peut, si les deux parents ont peu de recoupement entre eux, être une approche gagnante.

```
class B {
   // ...
};
class D0 : public B {
   // ...
};
class D1 : public B {
   // ...
};
class E : public D0, public D1 {
   // ...
};
```

Le choix d'illustrer les exemples à l'aide d'héritage public seulement est, tel que mentionné précédemment, un choix académique, orienté vers la simplicité. Plusieurs applications intéressantes de l'héritage multiple impliquent une combinaison plus riche et plus complexe de qualifications de sécurité.

Vous avez sans doute des questions quant à la nature de `B` dans `E`. Nous y reviendrons en long et en large plus loin dans cette section.

Héritage multiple et accès aux membres des parents

Imaginons les classes X et Y ci-dessous.

```
class X {
   int valeur_;
public:
   X (int valeur) : valeur_{valeur} {
   }
};
```

```
class Y {
   int valeur_;
public:
   Y (int valeur) : valeur_{valeur} {
   }
};
```

Imaginons maintenant la classe Z ci-dessous. *L'héritage appliqué étant public, tout* Z *est aussi un* X*, et tout* Z *est aussi un* Y*.*

Cette classe est correctement déclarée, et n'entraîne aucun conflit. Ses deux parents ont chacun un membre privé du même nom (l'attribut valeur_), mais puisque ces deux membres sont privés, aucun conflit ne résulte de la déclaration de Z.

```
struct Z : X, Y {
   Z(int val) : X{3 * val}, Y{4 * val} {
   }
};
```

Ici, en effet, une instance de la classe Z n'a accès ni à l'attribut X::valeur_, ni à l'attribut Y::valeur_, les deux étant privés pour leur classe respective.

Maintenant, compliquons un peu la sauce.

```
class X {
   int valeur_;
public:
   X (int valeur) : valeur_{valeur} {
   }
   int valeur() const noexcept {
      return valeur_;
   }
};
```

```
class Y {
   int valeur_;
public:
   Y (int valeur) : valeur_{valeur} {
   }
   int valeur() const noexcept {
      return valeur_;
   }
};
```

La classe Z semble ne pas avoir changé, or elle a maintenant deux membres publics du même nom (deux valeur()).

On peut la compiler et l'instancier sans peine, mais on ne peut pas appeler sa méthode valeur(), car ceci résulterait en une ambiguïté : devrait-on utiliser celle de son petit côté X ou celle de son petit côté Y?

```
struct Z : X, Y {
   Z (int valeur) : X{3 * val}, Y{4 * val} {
   }
};
```

Ce risque d'ambiguïté est possible pour tous les membres, attributs comme méthodes. De telles ambiguïtés ne se manifestent que si on essaie d'accéder au membre problématique[76]; la compilation d'un programme utilisant une classe dans laquelle un membre est ambigu se fait sans heurt si personne n'accède au membre en question de manière à générer un conflit.

Si on traite un Z comme un X (en passant un Z en paramètre à un sous-programme demandant un X), on pourra bien sûr utiliser sa méthode valeur() : ce sera alors, sans ambiguïté, son X::valeur().

La même situation et les mêmes conclusions s'appliquent au traitement d'un Z comme un Y.

[76] Dans notre exemple ici, tant que personne n'appellera la méthode valeur() à partir d'une instance de Z, il n'y aura aucun problème.

Mais si on veut traiter un `Z` comme un `Z`?

Dans une telle situation, la responsabilité de spécifier ce que signifie `valeur()` pour une instance de `Z` revient à la classe `Z` elle-même.

Par exemple, à droite, on aura choisi de définir la méthode `valeur()` d'un `Z` comme étant la somme des `valeur()` de ses parents.

C'est un choix arbitraire, bien sûr, mais en situation réelle, ce sont les problèmes à régler qui dicteront l'approche concrète à prendre.

```
struct Z : X, Y {
   Z(int val) : X{3 * val}, Y{4 * val} {
   }
   int valeur() const noexcept {
      return X::valeur() + Y::valeur();
   }
};
```

D'autres options sont proposées ci-dessous.

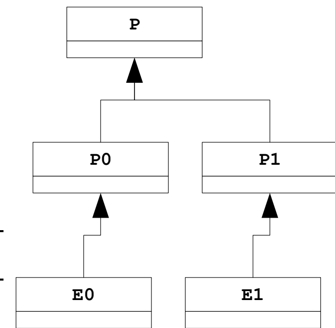
```
struct Z : X, Y {
   Z(int val) : X{3 * val}, Y{4 * val} {
   }
   // on enrobe X::valeur() (ou
   // Y::valeur()), simplement
   int valeur() const noexcept {
      return X::valeur();
   }
};
```

```
struct Z : X, Y {
   Z(int val) : X{3 * val}, Y{4 * val} {
   }
   // on défère valeur à la classe
   // X (ou à la classe Y), au choix.
   using X::valeur;
};
```

Question piège : quels seront les impacts si on choisit de spécifier la méthode `valeur()` de la classe `X` comme étant virtuelle et la méthode `valeur()` de la classe `Y` comme étant virtuelle? Vérifiez empiriquement votre hypothèse.

Question piège : quels seront les impacts si on choisit de spécifier la méthode `valeur()` de la classe `X` comme étant virtuelle tout en choisissant de ne pas spécifier la méthode `valeur()` de la classe `Y` comme étant virtuelle? Vérifiez empiriquement votre hypothèse.

Héritage virtuel



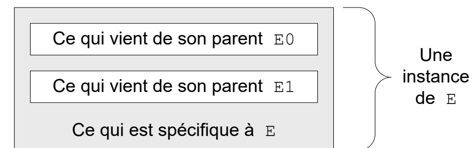
Imaginons les classes P, P0, E0, P1 et E1 visibles à droite. On a ici une situation d'héritage simple, avec par exemple P0 dérivant de P et E0 dérivant de P0. *Tout est légal*.

Que se produira-t-il si on désire ajouter la classe E qui dérive à la fois des classes E0 et E1?

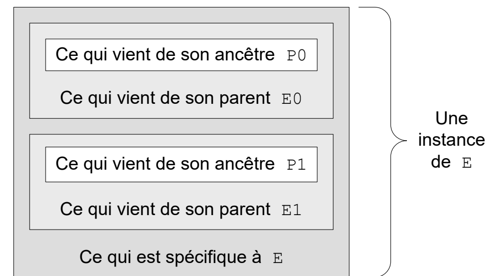
```
class E : public E0, public E1
```

Vous l'avez deviné: *E se retrouvera avec deux parties P dans son « arbre généalogique »*.

Pour le comprendre, examinons en détail la construction d'une instance de E.



Si on déclare e, une instance de E, et si on regarde e à la loupe, on verra que ce E est fait de son côté E0, puis de son côté E1, puis de ce qui lui est spécifique. L'ordre d'apparition des parents variera d'un compilateur à l'autre.

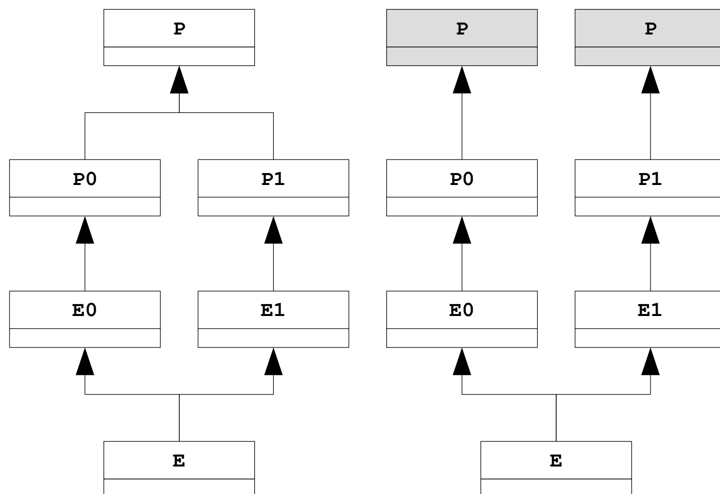


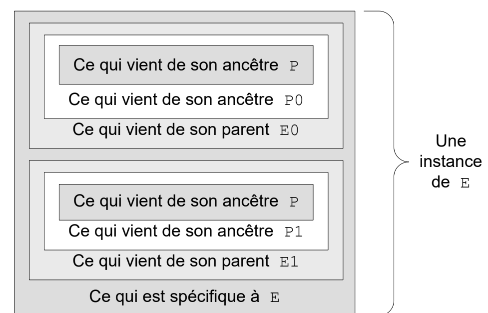
En regardant de plus près, on verra que la partie E0 d'un E inclut une partie P0, et on verra que la partie E1 inclut une partie P1.

Toujours en regardant de plus près, on constatera que les parties P0 et P1 incluent toutes deux leur propre partie P, chacune distincte de l'autre.

Il y aura donc, contrairement peut-être à l'intuition, deux P dans chaque E!

Là où, ci-dessous, on penserait avoir généré la hiérarchie à gauche, ***on a en fait généré celle à droite***.





Cette redondance de l'ancêtre P résulte en plusieurs problèmes potentiels, ou du moins en plusieurs considérations techniques.

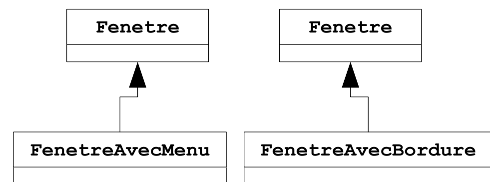
Par exemple :

- si le code client passe un E en paramètre à un sous-programme prenant un P, lequel des deux P dans ce E lui passera-t-il?
- si le code client réfère, dans un E, à un membre public d'un P (ce qui peut mener indirectement à un attribut privé d'un P), duquel des deux P parlera-t-il?

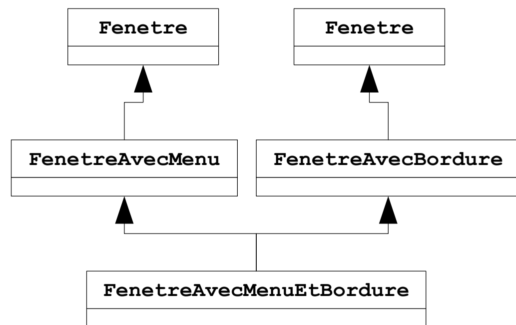
```
// classes P, P0, P1, E0, E1
// et E...
void f(P);
void g(E e) {
   // Ceci est-il légal? Si oui,
   // quel est son sens?
   f(e);
}
```

Les attributs des deux P peuvent avoir des valeurs différentes, étant distincts l'un de l'autre. Vous pouvez sans doute, à ce stade, imaginer sans peine les maux de tête pouvant résulter de telles situations. Pourtant, ces situations se produisent réellement dans des situations concrètes.

Par exemple[77], imaginons une classe Fenetre, et présumons que deux équipes de travail distinctes œuvrent en parallèle sur des classes qui en sont dérivées, soit FenetreAvecMenu et FenetreAvecBordure.



Présumons maintenant qu'on désire créer une classe FenetreAvecMenuEtBordure, dérivant par héritage multiple de FenetreAvecMenu et de FenetreAvecBordure. Du fait que ces deux parents ont été conçus à partir d'un ancêtre commun, on obtiendra le schéma visible à droite. Les membres de Fenetre seront dupliqués dans la classe terminale, ce qui peut ne pas être souhaitable ici.



C++ permet d'éviter ce genre de situation s'il ne s'agit pas là de l'effet désiré. On utilisera ce qu'on appelle ***l'héritage virtuel***.

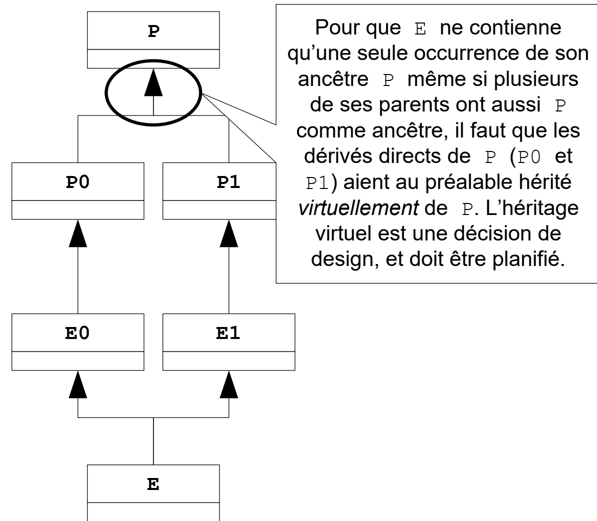
⇒ **L'héritage virtuel** permet la fusion des membres récurrents d'une hiérarchie de classes. Il n'est utile qu'en situation d'héritage multiple.

[77] L'exemple proposé ici vient de [StrouDE].

L'héritage virtuel s'implante en ajoutant la spécification **`virtual`** avant (ou après) la qualification de sécurité (`public` ou autre) qui précède le nom du parent lors de la déclaration d'une classe.

Il faut que toutes les classes souhaitant un parent fusionnel explicitent leur intention par l'héritage virtuel. Une seule ne suffit pas.

L'héritage virtuel est une décision de design, et doit être planifié dès le début. Pour le cas sous étude, il faut que les classes `P0` et `P1`, dérivés directs d'un ancêtre potentiellement redondant `P`, soient des dérivés virtuels de `P` pour que la fusion s'opère. On pourrait peut-être souhaiter que `E` puisse se sortir tout seul du pétrin, mais c'est malheureusement impossible dans ce cas-ci.



L'héritage virtuel soulève en effet beaucoup de questions de fond quant à la gestion des attributs d'un parent fusionnel (ici, les attributs de `P`) : quand seront-ils construits? Qui choisira le constructeur à invoquer? Quel sera l'ordre de destruction des classes de la hiérarchie? Pour ces raisons, on n'utilisera cette approche qu'une fois celle-ci planifiée à même l'ensemble des classes impliquées, pas seulement à partir des considérations de la classe terminale.

Les programmes ci-dessous illustrent l'un des impacts de l'héritage virtuel. Celui de gauche est illégal, dû à une ambiguïté lors de l'appel de `valeur()` dans `main()`, alors que celui de droite est légal : le parent immédiat de `P0` et de `P1` étant unique et fusionné dans un dérivé commun de `P0` et de `P1` (comme notre `E`), il n'y a qu'un seul `P` dans tout `E`.

Illégal

```
struct P {
   int valeur() const noexcept {
      return 3;
   }
};
class P0 : public P {};
class P1 : public P {};
class E0 : public P0 {};
class E1 : public P1 {};
class E : public E0, public E1 {};
int main() {
   E e;
   int i = e.valeur(); // ambigu
}
```

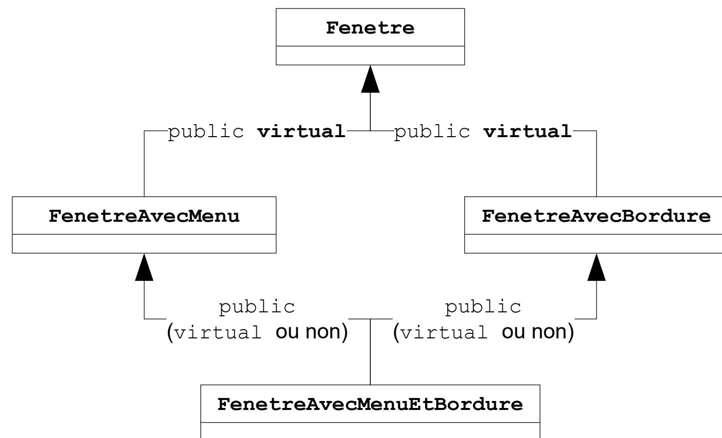
Légal

```
struct P {
   int valeur() const noexcept {
      return 3;
   }
};
class P0 : virtual public P {};
class P1 : virtual public P {};
class E0 : public P0 {};
class E1 : public P1 {};
class E : public E0, public E1 {};
int main () {
   E e;
   int i = e.valeur(); // non ambigu
}
```

Appliqué à notre exemple de fenêtres, plus haut, dériver virtuellement `FenetreAvecMenu` et `FenetreAvecBordure` de `Fenetre` nous mène au schéma ci-dessous, où une seule `Fenetre` apparaît dans la structure de la classe `FenetreAvecMenuEtBordure`.

Ce schéma est probablement plus près de ce qu'auraient eu en tête les concepteurs des différentes classes, dans la situation décrite ici.

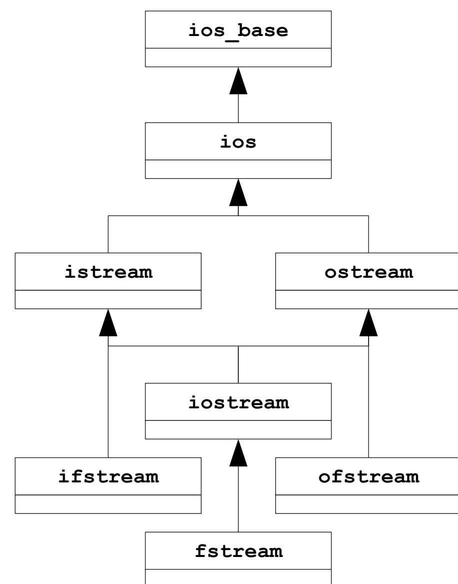
Évidemment, le design doit être guidé par les besoins d'entreprise et par les impératifs techniques : selon les projets et les classes, les deux designs (héritage virtuel ou non) peuvent avoir leur utilité.



Un cas d'héritage multiple (et virtuel) dont nous tirons profit sans cesse est celui des flux d'entrée/ sortie, incluant les classes `std::istream` et `std::ostream`, dont nous utilisons les instances bien connues que sont `std::cin` et `std::cout`.

Un extrait de la hiérarchie, telle qu'on la trouvera implantée dans certaines versions de la bibliothèque standard[78], se présente comme suit (toutes les classes indiquées ici sont dans l'espace nommé `std`, évidemment) :

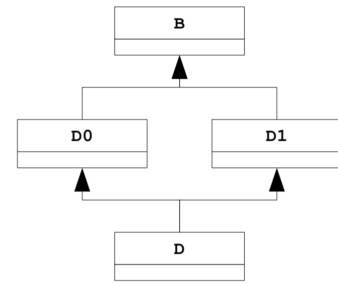
- la classe `ios_base` qui établit les types et les services de base de la hiérarchie;
- la classe `ios` qui définit ce que sont les flux d'entrée et de sortie;
- les classes `istream` et `ostream` qui en dérivent virtuellement et servent respectivement à titre de flux d'entrée et de sortie;
- la classe `iostream` qui spécifie un flux permettant à la fois les entrées et les sorties;
- les classes `ifstream` et `ofstream` qui servent respectivement à titre de flux d'entrée et de sortie sur des fichiers; et
- la classe `fstream` qui spécifie un flux permettant à la fois les entrées et les sorties sur un fichier.



L'applicabilité de l'héritage multiple et virtuel devrait être évidente à ce stade.

[78] Le portrait donné ici n'est qu'un survol incomplet mais se veut illustratif.

Héritage virtuel et cycle de vie



L'héritage virtuel soulève des questions délicates quant au cycle de vie des objets. Pour les illustrer, imaginons la situation suivante.

La classe `B` sert de parent virtuel aux dérivés `D0` et `D1`, et D dérive à la fois de `D0` et de `D1`. Il y a donc une seule occurrence de `B` dans un `D`.

On comprendra que construire `D` signifie construire sa partie `D0`, puis sa partie `D1` (ou l'inverse). On comprendra aussi que construire `D0` signifie construire `B`, et que construire `D1` signifie *aussi* construire `B`.

Le problème qui s'annonce est que `D0` existe en soi : cette classe a ses stratégies, ses conditions, ses intérêts et, si elle n'est pas abstraite, peut être instanciée par et pour elle-même. Les mêmes remarques s'appliquent d'ailleurs pour `D1`.

L'ébauche de code proposée à droite illustre cette réalité. Souvenez-vous que les membres et la qualification d'héritage d'un `struct` sont publics par défaut, ce qui explique l'écriture allégée. Notez que les appels aux constructeurs de `D0` et de `D1` dans `D` sont redondants mais ont été laissés dans le code à titre illustratif.

La question, donc, est de savoir quoi faire si la construction de `B` *via* `D0` entre en conflit avec celle de `B` *à travers* `D1`. Ou, plus simplement : *comment sera construit l'ancêtre fusionnel* `B` *dans un* `D`*?*

```
class B {
   int val_;
public:
   B(int val) : val_{val} {
   }
};
struct D0 : virtual B {
   D0() : B{3} {
   }
};
struct D1 : virtual B {
   D1() : B{4} {
   }
};
struct D : D0, D1 {
   D() : D0{}, D1{} {
   }
};
```

Si vous avez de la difficulté à saisir la problématique, alors imaginez ce qui se passerait si on avait une seule construction de `B` mais avec deux appels à son constructeur (un premier par la construction du `D0` dans `D`, et un autre par la construction du `D1` dans `D`). Avec la classe `D` à droite, l'attribut `B::val_` vaudrait-il alors `3` ou `4`?

Si vous avez un fond un peu pervers, imaginez la situation qui prévaudrait si l'attribut `val_` d'un `B` était une constante d'instance Risquerait-on une double initialisation de cette constante?

Et si le constructeur de `B` allouait dynamiquement de la mémoire, risquerait-on une double allocation mais (puisqu'il n'y aurait éventuellement qu'un seul appel au destructeur de `B`) une seule libération de mémoire?

Ces questions sont préoccupantes, et la liste n'est qu'embryonnaire.

Dénouer le nœud Gordien

Un choix doit être fait, peu importe lequel. On comprend mieux le choix fait en C++ à la lueur de ce que nous venons d'exposer :

- il est essentiel que le parent fusionnel (la partie parent virtuelle) ne soit construit qu'une seule fois, que cette construction précède celle de tous ses dérivés. Il est aussi essentiel que le parent fusionnel ne soit détruit qu'une seule fois, suite à la destruction de tous ses dérivés;

> Dans notre mise en situation, il importe que le constructeur et le destructeur de `B` ne soient invoqués qu'une seule fois puisqu'il n'y a qu'un seul `B` dans un `D`.

- il est essentiel que le parent fusionnel puisse exposer la gamme complète des constructeurs habituels, pour permettre (en fonction des besoins) à la fois la construction par défaut, par copie et diverses stratégies paramétriques;
- cependant, la seule entité capable de déterminer le bon constructeur à appeler pour un parent fusionnel est la classe terminale, soit l'enfant effectivement construit (ici : l'instance de `D`);
- en effet, si un programme instancie un dérivé intermédiaire (un `D0` par exemple), alors il est raisonnable d'estimer que le `D0` est l'entité la mieux placée pour choisir le constructeur à privilégier pour sa partie parent `B`. Dans le cas de la construction d'un `D`, il se trouve que le `D` lui-même une meilleure idée, une perspective plus claire du design et de la structure souhaitable pour le `B` que ses parents `D0` et `D1` ne le peuvent;

> En `POO`, rappelons-le, l'enfant connaît ses parents mais le parent ne connaît pas ses enfants.

- conséquemment, **dans un cas d'héritage virtuel, le constructeur à privilégier pour chaque parent fusionnel est choisi par la classe terminale**. En l'absence d'un choix explicite, le constructeur par défaut du parent fusionnel aura évidemment préséance.

```
D::D(const D &d)
   : B{d}, D0{d}, D1{d}
{
}
```

Ainsi, le constructeur par copie d'un `D` pourrait avoir l'air de ceci (à gauche). Bien que `B` soit un ancêtre de `D` et pas un parent immédiat, instancier un `D` fait en sorte de privilégier son choix de constructeur pour le `B` fusionnel car `D0` et `D1` dérivent virtuellement de `B`.

Ramifications de l'héritage virtuel

Si l'héritage virtuel évite la redondance des ancêtres dans une hiérarchie de classe, et si c'est le premier dérivé d'un ancêtre donné qui doit être un dérivé virtuel pour que cet ancêtre ne soit pas redondant (nous forçant ainsi à une grande prévoyance) alors *pourquoi ne pas faire de tout héritage un héritage virtuel*?

Sur le plan technique, une bonne raison est son coût en espace, en fait. Ce coût est minime[79], mais est là. Fidèle à sa position philosophique, C++ ne force aucune équipe de développement à faire, contre sa volonté, quelque chose qui impliquerait un coût en temps ou en espace, du moins pas s'il existe des situations où on pourrait s'en passer.

> En appliquant l'héritage virtuel, ce n'est pas le parent dont on dérive virtuellement qui grossira, mais bien chaque dérivé virtuel de ce parent. En effet, hériter virtuellement est une décision faite sur une base individuelle par et pour l'enfant. Le parent fusionnel, lui, n'en sait rien.

Sur le plan conceptuel, l'encapsulation vise entre autres à réduire le couplage entre les classes. Par exemple, les membres privés d'une classe ne sont pas connus des autres classes, pas mêmes de ses enfants immédiats. Réduire le savoir qu'une classe possède sur les autres facilite l'entretien du code; l'encapsulation participe à la réduction du couplage.

L'héritage multiple est aussi sain (sinon plus) que l'héritage simple pour ce qui est de l'assainissement des relations de couplage. L'héritage virtuel, par contre, force les enfants à connaître non seulement leurs parents immédiats, mais aussi certains de leurs ancêtres, et à contrôler la construction non pas de leur petite partie d'une hiérarchie mais bien de plusieurs parties d'eux-mêmes.

Pour cette raison, il est sage d'éviter les hiérarchies virtuelles trop complexes. La gestion et l'entretien de ces structures coûte cher. Comme toujours, vaut mieux choisir les bonnes armes pour chaque combat.

Héritage multiple et affectation par défaut

Le standard officiel de C++ définit l'ordre de construction d'un objet dans une structure d'héritage virtuel en laissant l'objet le plus dérivé de la hiérarchie déterminer l'ordre de construction des bases fusionnelles. En retour, le standard ne définit pas avec précision l'ordre dans lequel sont faites les affectations des parents lorsqu'un programme réalise une copie par affectation pour un tel enfant. La plupart des compilateurs offrent une stratégie ou l'autre (par exemple affecter les ancêtres du plus haut au plus bas dans la hiérarchie), mais le comportement par défaut risque d'être non portable d'un compilateur à l'autre.

Il est donc sage d'implémenter explicitement l'affectation dans un objet à l'intérieur duquel certaines bases sont fusionnelles.

---

[79] Un pointeur de plus dans toute instance d'une classe ayant un parent virtuel, en fait. Le coût est le même que pour l'ajout d'au moins une méthode virtuelle dans une classe, alors un dérivé virtuel muni d'au moins une méthode virtuelle occupe un espace identique à la même classe sans héritage virtuel ou méthodes polymorphiques, plus l'espace requis pour entreposer deux pointeurs.

Applications de l’héritage multiple

L’une des principales applications de l’héritage multiple, utile à un point tel qu’elle est appliquée à titre de concept à part dans la plupart des langages qui ne tolèrent que l’héritage simple, est ce qu’on nomme l’***héritage d’interfaces***, par opposition à l’*héritage d’implémentation*.

Comme bien des idées abstraites, celle-ci se saisit mieux à partir d’un exemple. Ainsi, supposons qu’on vise à construire une hiérarchie servant à représenter des animaux.

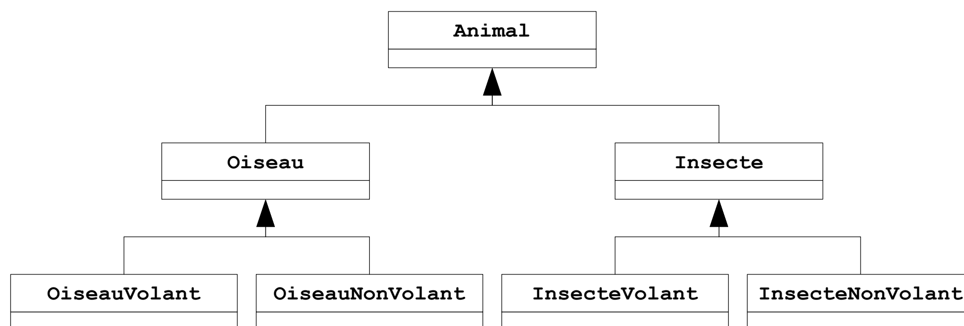
On positionnerait peut-être la classe `Animal` tout en haut de notre arbre, pour qu’elle contienne les membres que tout animal doit avoir (un poids; une taille; des mutateurs et accesseurs pour ces attributs; ce genre de chose).

Supposons ensuite qu’on ait deux classes dérivées d’`Animal`, soit `Oiseau` et `Insecte`. Les oiseaux ont deux pattes et des plumes, alors que les insectes ont six pattes et une carapace[80].

Remarquez :

- certains oiseaux volent, comme le colibri. D’autres ne volent pas, comme l’émeu;
- certains insectes volent, comme le papillon. D’autres ne volent pas, comme le cafard.

Ceci nous mène naturellement au schéma suivant :



[80] Pour faire simple; entendu ici que ce cours n’est *clairement* pas un cours de zoologie ou d’entomologie.

Héritage d'implémentation

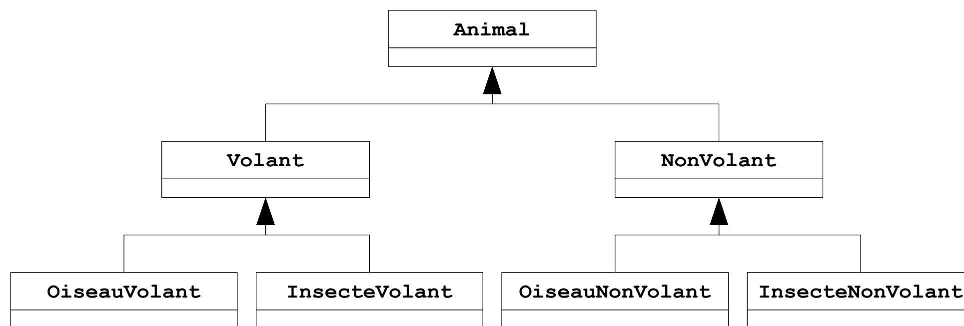
Notre hiérarchie ci-dessus est basée sur l'héritage d'implémentation. Nous avons posé des choix structurels qui reposent sur la constitution interne des objets à représenter. Ceci permet, à chaque étape dans notre chemin de spécialisation progressive, de bâtir les enfants à partir des états définis pour leurs ancêtres sans avoir à le définir à nouveau.

> L'idéal visé ici est la *réutilisation des états*. Le choix de découper en fonction des états est un choix philosophique parmi plusieurs choix possibles; ce n'est pas le seul choix possible.

Par contre, cette approche a ses défauts. Par exemple, si on veut traiter tous les animaux volants comme tels, par exemple pour profiter d'opérations communes à tous les volants comme `decoller()`, `voler()` et `atterrir()`, notre hiérarchie fait un peu défaut. Le design choisi, en visant l'héritage d'implémentation, manque la cible pour ce qui est de la généralisation polymorphique des opérations.

Dans le cas exposé ci-dessus, à moins d'implanter ces opérations à même `Animal` (ce que nous ne voulons pas nécessairement faire, entendons-nous[81]), nous ne sommes pas en mesure de traiter sur un pied d'égalité *opérationnel* un papillon et un colibri.

Si nous avions classé les animaux d'abord en les subdivisant entre volants et non volants, on aurait obtenu une hiérarchie différente, plus utile pour régler le problème du déplacement d'un animal, mais découlant probablement sur un problème similaire du fait que (pour ne donner qu'un exemple) les oiseaux ne seraient plus regroupés entre eux :



L'héritage simple à lui seul, orienté surtout vers l'héritage d'implémentation, fait en sorte que tout choix hiérarchique devienne un compromis imparfait. La décision structurelle nuit au volet opérationnel, et la solution opérationnelle introduit de la redondance dans l'écriture de code.

Plus concrètement, dans les deux cas, nous obtenons les mêmes classes terminales, mais avec des déficiences distinctes. La version regroupant tous les oiseaux sous un même ancêtre réduit le code à rédiger pour un oiseau donné, mais empêche de traiter tous les volants comme des volants, ceux-ci n'ayant pas d'ancêtre commun pour servir d'abstraction polymorphique en vue du concept de volant. La version regroupant tous les volants sous un même ancêtre permet de traiter par polymorphisme tous les volants comme des volants, mais il n'est plus possible de profiter du code commun à tous les oiseaux dans l'implémentation d'un oiseau spécifique (à moins de faire des manœuvres par composition et par délégation, ce qui manque un peu d'élégance).

[81] Ce serait un poids inutile à porter pour la classe `Baleine`, pour ne nommer que celle-là…

Héritage d'interfaces

Imaginons maintenant que nous découpions le problème autrement. Notre problème tient au fait que nous avons en fait deux angles d'approche pour découper nos classes : un angle structurel, mettant l'accent sur les similitudes du point de vue des états, et un angle opératoire, mettant l'accent sur le polymorphisme et la généralisation. La voie royale est, clairement, de produire non pas une hiérarchie, mais bien deux hiérarchies!

Commençons par concevoir une classe abstraite nommée `Volant` qui dévoile les méthodes abstraites `decoller()`, `voler()` et `atterrir()`. Tout `Volant` saura voler, atterrir et décoller à sa manière. Cette classe sera la racine de notre hiérarchie opératoire, tout comme `Animal` est la racine de notre hiérarchie structurelle.

Mettons aussi au point les classes `Oiseau` et `Insecte`, raffinements de la classe `Animal`, pour regrouper structurellement ce qui est considéré commun à tous les oiseaux ou à tous les insectes, respectivement.

Ensuite, procédons à la création d'une classe dérivée à la fois d'`Insecte` et de `Volant`, classe que nous nommerons (pour concorder avec les exemples précédents) `InsecteVolant`, et faisons de même pour `OiseauVolant` qui dérivera d'`Oiseau` et de `Volant`. La classe `InsecteVolant` devra implanter les méthodes de `Volant`, et il en ira de même pour la classe `OiseauVolant`.

⇒ Créer une classe de base abstraite comme `Volant` dans le but de regrouper des classes par souci opérationnel est un exemple direct de ce qu'on entend par **héritage d'interfaces**.

Le design par héritage simple d'implémentation et par héritage multiple d'interfaces, rendu possible dans la majorité des langages `OO` pragmatiques, est applicable chaque fois qu'un découpage hiérarchique complexe implique une seule branche structurelle et autant de branches opératoires que nécessaire.

Ayant appliqué, en surimpression et par héritage multiple, la classe `Volant` à deux classes normalement disjointes de notre hiérarchie, nous venons de nous doter de la capacité de les traiter toutes deux comme des cas particuliers de `Volant`, et de la capacité de manipuler divers dérivés de `Volant` par polymorphisme.

> L'héritage d'interfaces est une application répandue de l'héritage multiple. La réciproque n'est pas vraie : il existe des situations d'héritage multiple qui ne sont pas des situations d'héritage d'interfaces. Plusieurs langages, dont Java et les langages `.NET` font le choix de ne permettre que l'héritage d'interfaces au sens le plus strict : le mot **`interface`** y est un mot clé pour une classe abstraite sans le moindre attribut et ne contenant que des méthodes abstraites. L'interface est alors présentée comme un concept distinct de celui de classe. Une classe peut, dans ces langages, n'avoir qu'un seul parent mais implémenter autant d'interfaces que désiré. Ce choix des concepteurs force à réécrire fréquemment le même code de manière redondante dans plusieurs classes distinctes, ou à appliquer fréquemment une approche composition/ délégation.
>
> Avec C++, les interfaces seront souvent des `struct` et seront munies d'un destructeur polymorphique.

Hiérarchies planes

Un effet sympathique de l'héritage multiple est qu'il permet d'éviter les hiérarchies très profondes et très verticales qu'on peut trouver entre autres dans les bibliothèques de classes comme l'infrastructure de `.NET` ou la bibliothèque de classes de Java. Les hiérarchies verticales sont des arbres dont les classes feuilles accumulent presque inévitablement du bagage qui leur est inutile. Les instancier donne accès à une multitude de services… dont plusieurs y sont superflus. Les objets tendent alors à être trop gros, ce qui ne change pas grand-chose dans les petits systèmes mais peut être très coûteux dans un système commercial complexe.

Les hiérarchies horizontales, elles, sont faites en combinant dans une classe enfant des classes qui sont plutôt minimalistes et complètes en elles-mêmes. Un enfant deviendra un assemblage de plusieurs parents *a priori* indépendants les uns des autres, de manière semblable à une approche par composition. Cela tend à réduire le surplus de code et de données chez les enfants.

L'héritage multiple (ou, dans une moindre mesure[82], l'héritage simple d'implémentation combiné à l'héritage multiple d'interfaces) est un outil précieux pour aplanir les hiérarchies de classes. En déplaçant vers les classes terminales le choix de combiner les parents, l'approche naturelle devient de développer plusieurs petites classes disjointes entre elles et à les combiner de manière tardive, selon les besoins. Le couplage entre les classes est alors très faible, ce qui accroît de manière importante la cohérence (une classe, une vocation) et la facilité avec laquelle sont réalisés les tests et l'entretien.

Combiner les stratégies

Dans une entrevue[83] avec les concepteurs de C (**Dennis Ritchie**), C++ (**Bjarne Stroustrup**) et Java (**James Gosling**), Stroustrup offre une application pragmatique et simple de l'héritage multiple : si des structures de données et des opérations forment un tout cohérent, il est possible de les combiner dans une classe (appelons-la `Commun` pour faire simple et général, bien que ce soit là un mauvais nom pour du code de production) qui ne serait offerte qu'aux gens appelés à développer avec leur aide.

Une autre classe (nommons-la `Interface`, pour des raisons évidentes, mais elle pourrait contenir des attributs et des méthodes non abstraites si le langage le permet; encore une fois, ce nom est trop générique pour du code de production) pourrait, elle, exposer des services visibles à tout utilisateur du produit en cours de développement.

[82] … parce que cela implique alors de travailler fréquemment à l'aide d'une approche enrobage/ délégation, là où l'héritage privé ou protégé aurait allégé l'écriture.

[83] Voir http://www.gotw.ca/publications/c_family_interview.htm. Ça date de l'an `2000`, mais il y a plusieurs réflexions là-dedans sur la programmation, l'enseignement, l'industrie, *etc*.

On aurait donc deux blocs fondamentaux :

Pour usage interne seulement	Livré aux utilisateurs

`class Commun {`
`   // attributs, méthodes qui ne sont connus`
`   // que des développeurs de l'outil`
`};`
	`class Interface {`
	`   // services pour les utilisateurs`
	`   // (face visible de l'outil développé)`
	`   virtual ~Interface() = default;`
	`};`

La conception d'outils à partir de ces blocs pourrait prendre plusieurs tangentes. Deux développeurs différents pourraient livrer leur propre version du même outil : l'un pourrait concevoir une version très rapide mais coûteuse en mémoire, et l'autre une version économique mais un peu moins rapide.

Puisque les utilisateurs de l'outil ne verront que le volet public de l'outil (`Interface`, qui est abstraite) et ne l'utiliseront, par définition, que par polymorphisme, le code utilisateur pourra utiliser l'une ou l'autre des implémentations de l'outil sans devoir être modifié de manière fondamentale (au pire, en changeant la ligne servant à instancier l'outil).

```
class ImplementationRapide
   : public Interface, protected Commun
{
   // ...
};
```

```
class ImplementationEconomique
   : public Interface, protected Commun
{
   // ...
};
```

C'est là une application simple et utile de l'héritage multiple, qui permet aux développeurs de partager harmonieusement une base de code pertinente et qui permet aux utilisateurs de profiter de l'encapsulation et de l'abstraction du modèle OO pour concevoir des programmes résistants aux changements d'implémentation.

```
// return new ImplementationRapide;
Interface *creerRapide();
// return new ImplementationEconomique;
Interface *creerEcono();
int main() {
   auto p = creerRapide();
   // ...utiliser...
   delete p;
}
```

Parenthèse contemporaine

Il est malsain de faire retourner un pointeur brut d'une fonction comme le fait `creerRapide()` ci-dessus; cela rend le code client responsable de libérer ultérieurement le pointé, et encourage les fuites de mémoire.

Nous verrons en détail, dans [POOv02], une solution à ce problème, soit le recours à un pointeur intelligent, sorte d'objet se comportant syntaxiquement comme un pointeur mais étant responsable de son pointé. Ce qui suit n'est qu'un bref avant-goût des pratiques contemporaines que nous explorerons alors, et n'a rien à voir vraiment avec l'héritage multiple (on parle vraiment ici de saines pratiques de programmation).

Pour que le portrait soit simple, reprenons les implémentations ci-dessus, et supposons (pour alléger l'exemple) que les classes aient des constructeurs par défaut publics :

```
class ImplementationRapide
   : public Interface, protected Commun
{
   // ...
public:
   ImplementationRapide();
   // ...
};
```

```
class ImplementationEconomique
   : public Interface, protected Commun
{
   // ...
public:
   ImplementationEconomique();
   // ...
};
```

Étant donné le code existant, les versions à droite de `creerRapide()` et de `creerEcono()` retournent maintenant non pas des pointeurs bruts mais bien des pointeurs intelligents.

Le programme principal obtient un `unique_ptr<Interface>` (j'ai écrit `auto` par souci d'économie) nommé `p`.

Par `p`, il a accès à tous les services du type `Interface`. Parce que p est un pointeur intelligent, `main()` n'a pas à libérer le pointé manuellement; en effet, à la mort de `p`, le pointé qu'est `*p` sera automatiquement libéré.

```
// ...
#include <memory>
std::unique_ptr<Interface> creerRapide() {
   return make_unique<ImplementationRapide>();
}
std::unique_ptr<Interface> creerEcono() {
   return make_unique<ImplementationEconomique>();
}
int main() {
   auto p = creerRapide();
   // ...utiliser...
}
```

En résumé

On pensera l'héritage d'implémentation sous forme *généalogique*. C'est là l'héritage le plus classique, celui qu'on entend par l'énoncé *hériter, c'est être tout ce que notre parent est, et un peu plus*. Chaque nouvel étage d'une hiérarchie ajoute aux états et aux opérations d'un objet.

On pensera l'héritage d'interface sous forme *opératoire*. Plutôt que d'hériter de code existant et de données existantes, *on héritera de l'opportunité (*ou, pour les abstractions pures, *de l'obligation) d'implanter certaines méthodes* (d'implanter une interface, en fait) pour permettre le polymorphisme entre classes partageant entre elles certaines fonctionnalités.

Les hiérarchies plus horizontales ont, en général, un plus faible couplage que les hiérarchies verticales. Ceci simplifie les tests et l'entretien du code.

Dans d'autres langages

Nous pénétrons dans un espace miné en abordant la question de l'héritage multiple dans d'autres langages, puisque cette question n'est pas que source de *différences* entre langages mais aussi source de *différends* entre les tenants des positions philosophiques sous-tendues par chacun de ces outils. Nous frappons ici un cas où le concept d'un langage (Java, C#, `VB.NET`) est une technique dans un autre langage (comme C++), un peu comme les objets constants (qui existent en C++ et se simulent en Java, C# ou `VB.NET` en construisant des classes dont les instances sont immuables)[84].

La disponibilité et l'application d'héritage multiple d'implémentation ou d'interfaces est l'un des facteurs influençant le plus la conception `OO`. Porteur d'un paradigme à part entière, ce sujet entraîne autant de discussions que ne le fait la dichotomie entre langages moussant les décisions statiques et langages mettant l'accent sur les décisions dynamiques[85] ou encore la dichotomie entre les langages hautement dynamiques (souvent des langages de scripts comme Ruby ou Python) et langages mettant l'accent sur une étape de compilation, que ce soit pour une machine réelle ou virtuelle, ce qui inclut les langages Java, C#, `VB.NET` ou C++.

Les langages Java, C# et `VB.NET` partagent tous trois une racine conceptuelle commune n'acceptant que l'héritage simple d'implémentation mais permettant l'héritage multiple d'interfaces. Dans les trois cas, l'acceptation de cette dichotomie comme concept entraîne l'introduction d'un mot clé (le mot **`interface`**) et entraîne aussi dans deux cas sur trois une distinction entre hériter d'un parent (**`extends`** en Java, **`Inherits`** en `VB.NET`) et implémenter une interface (mot clé **`implements`**), bien qu'implémenter les méthodes d'une interface ou d'une classe parent abstraite soit, sur le plan technique, exactement la même chose.

Chose certaine : il semble y avoir, de manière universelle, reconnaissance de l'importance de supporter l'héritage d'interfaces en plus de l'héritage d'implémentation (l'héritage au sens traditionnel, ce qui signifie dans bien des langages l'héritage à proprement dit).

Tel que mentionné plus haut dans la section ***Héritage d'interfaces***, le seul héritage simple d'implémentation ne permet pas de représenter tous les points de vue légitimes sur une organisation hiérarchique de classes. Un design solide demande au minimum qu'on puisse y apposer plusieurs points de vue; ne pas permettre l'héritage multiple est une chose, mais empêcher aussi l'héritage d'interfaces tue le design.

Les langages `.NET` supportent aussi un cousin germain *ad hoc* de l'héritage d'interfaces : les **délégués** (`delegate`), qui permettent de réaliser du polymorphisme sur la base de la signature d'une méthode plutôt que sur la base d'un support structurel commun et constituent un hybride entre les pointeurs de fonctions de C et de C++ et certaines applications de l'instanciation au besoin de classes anonymes en Java.

[84] Voir à ce sujet ***Simuler l'héritage multiple***, plus loin.

[85] Classification qui place parfois les tenants de Java ou des langages `.NET` en confrontation philosophique avec les *aficionados* de C++.

En C++ tout comme (à un degré *beaucoup* moindre) en Java et en C#, la **programmation générique** [POOv02] permet aussi de résoudre certains problèmes pour lesquels une approche structurelle comme l'héritage multiple ou l'héritage d'interfaces pourraient rendre service.

La principale raison d'être des interfaces (au sens OO exprimé par le mot clé interface) est de permettre du polymorphisme selon plusieurs points de vue. Conceptuellement, les méthodes d'une interface prise au sens strict sont nécessairement des méthodes d'instance publiques et abstraites (donc à la fois polymorphiques et devant absolument être surchargées). En ce sens, il est possible de conceptualiser l'idée d'interface OO en C++ en utilisant des struct (ou des class dont les méthodes sont toutes publiques) ne contenant que des méthodes abstraites.

En Java, une interface se déclare à l'aide du mot clé interface. Une classe dérivant d'une interface (ou, en termes Java, implémentant une interface) l'indiquera à l'aide du mot clé implements. Une interface peut par contre dériver d'une autre interface en utilisant le mot clé extends. Une classe ne peut quant à elle utiliser extends que pour décrire une relation d'héritage simple d'implémentation.

Remarquez, à droite, l'appel à la méthode Z.présenter() passant en paramètre une instance de la classe CouleurPrésentable là où un Présentable est requis. Clairement, il s'agit d'un cas de polymorphisme.

```
interface Présentable {
   String getPrésentation();
}
class Couleur {
   public enum Valeur {
      Rouge, Vert, Bleu
   };
   private Valeur valeur;
   public Couleur(Valeur val) {
      setValeur(val);
   }
   private void setValeur(Valeur val) {
      valeur = val;
   }
   public Valeur getValeur() {
      return valeur;
   }
}
class CouleurPrésentable
   extends Couleur implements Présentable {
   public CouleurPrésentable(Valeur val) {
      super(val);
   }
   public String getPrésentation() {
      return "Je suis " + getValeur();
   }
}
public class Z {
   // polymorphisme par interface
   private static void présenter(Présentable p) {
      System.out.println(p.getPrésentation());
   }
   public static void main(String [] args) {
      CouleurPrésentable cp=
         new CouleurPrésentable(Couleur.Valeur.Vert);
      présenter(cp);
   }
}
```

En C#, la syntaxe applicable aux cas d'héritage d'interfaces est un mélange de celles de C++ et de Java. Le mot clé `interface` déclare une interface, abstraction polymorphique ne contenant que des méthodes d'instance publiques et abstraites.

Comme en Java, la qualification `public` est omise pour les méthodes des interfaces parce qu'elle est redondante.

Une classe déclare dériver d'une autre classe comme elle déclare implémenter une interface : en ayant recours au symbole **:** suivi du nom de la classe parent et de la liste des interfaces qui seront implémentées. Le symbole pour délimiter le nom des parents (interfaces incluses) est la virgule.

```
namespace z
{
   interface Présentable
   {
      string GetPrésentation();
   }
   class Couleur
   {
      public enum Valeur { Rouge, Vert, Bleu }
      public Valeur Val { get; private set; }
      public Couleur(Valeur val) { Val = val; }
   }
   // héritage simple d'implémentation +
   // héritage d'interface
   class CouleurPrésentable: Couleur, Présentable
   {
      public CouleurPrésentable(Valeur val)
         : base(val)
      {
      }
      public string GetPrésentation()
      {
         return "Je suis " + Val;
      }
   }
   public class Z
   {
      // polymorphisme par interface
      private static void Présenter(Présentable p)
      {
         System.Console.WriteLine(p.GetPrésentation());
      }
      public static void Main(string [] args)
      {
         CouleurPrésentable cp =
            new CouleurPrésentable(Couleur.Valeur.Vert);
         Présenter(cp);
      }
   }
}
```

En VB.NET, le code suit comme à l'habitude un mélange des règles de C# et de la syntaxe des langages de la lignée VB.

Remarquez un détail technique n'ayant rien à voir avec les concepts explorés ici : la chaîne retournée par la méthode GetPrésentation() de CouleurPrésentable est construite à l'aide de la méthode de classe Format() de la classe String. Ceci est nécessaire du fait que la concaténation typique de chaînes de caractères VB (avec l'opérateur &) appliqué à une entité énumérée ne conserve que la valeur entière associée à cette entité (ici : 1 plutôt que Vert) ce qui rendrait l'affichage produit par cet exemple incohérent avec celui des exemples C# et Java, plus haut.

```
Namespace z
   Interface Présentable
      Function GetPrésentation() As String
   End Interface
   Class Couleur
      Public Enum Valeur
         Rouge
         Vert
         Bleu
      End Enum
      Private valeur_ As Valeur
      Public Sub New(ByVal val As Valeur)
         SetValeur(val)
      End Sub
      Private Sub SetValeur(ByVal val As Valeur)
         valeur_ = val
      End Sub
      Public Function valeur() As Valeur
         valeur = valeur_
      End Function
   End Class
   ' héritage simple d'implémentation + héritage d'interface
   Class CouleurPrésentable
      Inherits Couleur
      Implements Présentable
      Public Sub New(ByVal val As Valeur)
         MyBase.New(val)
      End Sub
      Public Function GetPrésentation() As String _
         Implements Présentable.GetPrésentation
         Dim val As Valeur = valeur()
         GetPrésentation = String.Format("Je suis {0}", val)
      End Function
   End Class
   Public Class Z
      ' polymorphisme par interface
      Private Shared Sub Présenter(ByVal p As Présentable)
         System.Console.WriteLine(p.GetPrésentation())
      End Sub
      Public Shared Sub Main()
         Dim cp As CouleurPrésentable = _
            New CouleurPrésentable(Couleur.Valeur.Vert)
         Présenter(cp)
      End Sub
   End Class
End Namespace
```

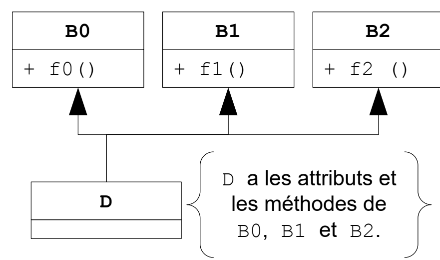
Simuler l'héritage multiple

Certains langages OO, dont Java et les langages .NET, supportent l'héritage multiple d'interfaces mais ne supportent pas l'héritage multiple d'implémentation. Une interface au sens Java et C# équivaut à une classe abstraite C++ qui ne posséderait que des méthodes abstraites (aucun attribut, aucune méthode qui serait munie d'une définition).

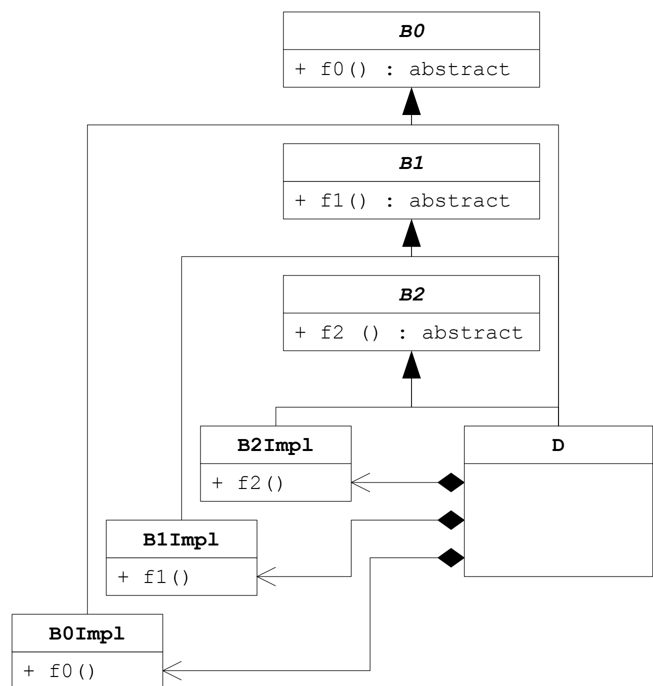
Il advient, évidemment, que le besoin d'héritage multiple d'implémentation se fasse sentir même dans des langages où seul l'héritage multiple d'interfaces est permis. Pour en arriver à un résultat semblable à celui obtenu par héritage multiple, il faut alors travailler un peu plus fort. La recette va comme suit :

- rassembler les opérations de chaque parent dans une interface, donc dans une classe strictement abstraite;
- créer une classe dérivant de chaque interface (de chaque classe strictement abstraite) et implémentant chaque méthode comme bon nous semble, de la manière la plus standard possible, comme l'aurait fait le parent dans une situation d'héritage multiple; puis
- faire en sorte que la classe terminale dérive de chaque interface, y insérer un attribut de chaque classe implémentant les fonctionnalités par défaut d'un parent, et déléguer les fonctionnalités en question de l'enfant vers la classe simulant le parent.

Le diagramme de classes proposé à droite et présentant la relation d'héritage multiple par laquelle D dérive de B0, B1 et B2 équivaut, si on simule l'héritage multiple avec une paire composition/héritage d'interfaces, au diagramme proposé un peu plus bas.



- D dérive de B0, B1 et B2, toutes trois abstraites (toutes trois des interfaces strictes). D s'engage donc à coder les méthodes de ces trois interfaces;
- les classes B0Impl, B1Impl et B2Impl implémentent respectivement de manière standard les méthodes de B0, B1 et B2. Ce sont des *classes d'implémentation partielle*;
- D inclut par composition un B0Impl, un B1Impl et un B2Impl. L'implémentation que fait D des méthodes de B0, B1 et B2 est de les déléguer à la bonne classe d'implémentation partielle.



Au prix d'une hausse de complexité, on parvient ainsi au même niveau d'opérabilité que pour une situation d'héritage multiple.

Classes imbriquées

```
class Externe {
   // ...
   class Interne {
      // ...
   };
   // ...
};
```

On peut déclarer une classe qui soit interne à une autre classe, ce que nous faisons régulièrement pour les classes représentant des exceptions. Une telle classe est parfois nommée classe interne (à une autre classe), et est parfois nommée classe imbriquée. Une classe imbriquée peut être privée, protégée ou publique, comme n'importe quel autre membre, ce qui influence la capacité qu'a (ou non) le code client de lui accéder.

Une classe interne a directement accès à tous les membres de classe de sa classe externe, peu importe leur qualification de sécurité. Elle fait partie de la classe dans laquelle elle est déclarée, au même titre que les méthodes qui y sont déclarées.

On peut référer directement à une classe interne à l'aide d'une syntaxe rappelant celle des membres de classe. Par exemple, si la classe `Interne` est interne à la classe `Externe`, comme dans l'extrait ci-dessus, alors son nom complet sera `Externe::Interne`.

En général, les classes imbriquées seront un choix de design intéressant lorsque :

- la classe interne n'a de sens que dans le contexte de la classe externe;
- la classe interne n'a de sens que si la classe externe est disponible;
- la classe interne est une sorte de service exposé par la classe externe; ou encore
- la classe interne est vouée à servir d'outil de gestion interne pour la classe externe.

Exemple d'utilisation de classe imbriquée

Vous trouverez ci-dessous un exemple de classe imbriquée. Cet exemple vous propose une classe Resultats, qui garde en note des résultats scolaires, qu'il s'agisse de ceux d'un(e) étudiant(e) ou d'une classe entière, peu importe.

Pour faciliter la bonne gestion des notes sur une base individuelle, une classe Note est déclarée de manière interne et privée à Resultats. Cette classe assure la validité de chaque note prise sur une base individuelle. Chaque Note sait si elle représente une réussite ou un échec.

La classe Resultats permet d'ajouter des notes à celles déjà emmagasinées. Les instances de Note conservées par une instance de Resultats sont en fait placées dans un vecteur standard. La classe Resultats offre des services de calcul de la moyenne, du nombre d'échecs et du nombre de réussites pour les résultats qu'elle représente.

```
#ifndef RESULTATS_H
#define RESULTATS_H
// Resultats.h
#include <vector>
#include <iosfwd>
class Resultats {
   class Note { // classe interne privée
   public:
      using value_type = float;
      class Invalide {}; // classe interne à la classe interne!
   private:
      static const value_type
         MIN_VALEUR, MAX_VALEUR, SEUIL_PASSAGE;
      value_type valeur_ = MIN_VALEUR;
      static value_type valider(value_type val) {
         if (val < MIN_VALEUR || MAX_VALEUR < val) throw Invalide{};
         return val;
      }
   public:
      value_type valeur() const noexcept {
         return valeur_;
      }
      Note()= default;
      Note(value_type valeur) : valeur_{valider(valeur)} {
      }
      bool operator==(const Note &n)const noexcept {
         return valeur() == n.valeur();
      }
      bool operator!=(const Note &n) const noexcept {
         return !(*this == n);
      }
      bool operator<(const Note &n) const noexcept {
         return valeur() < n.valeur();
```

```
      }
      bool operator<=(const Note &n) const noexcept {
         return !(n < *this);
      }
      bool operator>(const Note &n) const noexcept {
         return n < *this;
      }
      bool operator>=(const Note &n) const noexcept {
         return !(*this < n);
      }
      bool reussite() const noexcept {
         return *this >= SEUIL_PASSAGE;
      }
      bool echec() const noexcept {
         return !reussite();
      }
   };
   // Vecteur de Resultat::Note (on peut omettre « Resultat:: », étant dans Resultat)
   std::vector<Note> resultats_;
public:
   using value_type = Note;
   using size_type = std::vector<Note>::size_type;
   Resultats() = default;
   template <class It>
      Resultats(It debut, It fin) { // constructeur de séquence
         for(; debut != fin; ++debut)
            ajouter(*debut);
      }
   void ajouter(const Note &n) {
       resultats_.push_back(n);
   }
   class Vide {}; // classe interne!
   value_type moyenne() const;
   size_type nb_echecs() const noexcept;
   size_type nb_reussites() const noexcept;
   size_type nb_resultats() const noexcept {
      return resultats_.size();
   }
   bool vide() const noexcept {
      return resultats_.empty();
   }
};
std::ostream& operator<<( std::ostream&, const Resultats::value_type&);
#endif
```

Remarquez que les classes `Resultats` et `Resultats::Note` sont des types valeurs, ce qui allège leur écriture et leur manipulation. Bien que `Resultats::Note` soit une interne et privé pour `Resultats`, `Resultats::value_type` le révèle en partie.

Pour ce qui est de l'implémentation, maintenant :

```
// Resultats.cpp
#include "Resultats.h"
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;
ostream& operator<<(ostream &os, const Resultats::value_type &n) {
   return os << n.valeur();
}
auto Resultats::moyenne() const -> value_type {
   if (vide()) throw Vide{};
   Note::value_type somme = {};
   // mieux : l'algorithme standard std::accumulate... essayez-le!
   for (auto & res : resultats_)
      somme += res.valeur();
   return somme / nb_resultats();
}
auto Resultats::nb_echecs() const noexcept -> size_type {
   auto n = size_type{};
   // mieux : l'algorithme standard std::count_if... essayez-le!
   for (auto & res : resultats_)
      if (res.echec())
         ++n;
   return n;
}
auto Resultats::nb_reussites() const noexcept -> size_type {
   return nb_resultats() - nb_echecs();
}
// Initialisation des constantes de classe de la classe interne Resultats::Note
const Resultats::Note::value_type
   Resultats::Note::MIN_VALEUR    =   0,
   Resultats::Note::MAX_VALEUR    = 100,
   Resultats::Note::SEUIL_PASSAGE =  60;
```

Si vous en avez envie, les algorithmes de la bibliothèque standard `<algorithm>` que sont `accumulate()` et `count_if()` permettent d'accélérer et d'alléger l'implémentation proposée ci-dessus.

Enfin, à titre d'exemple, voici un programme de test pour la classe `Resultats`. La classe interne `Resultats::Note` y est invisible.

```
#include "Resultats.h"
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
   Resultats r{30, 80, 78.5, 12, 60, 61, 99};
   cout << "Moyenne: "    << r.moyenne() << ", "
        << "Nb. echecs: " << r.nb_echecs() << ", "
        << "Nb. succes: " << r.nb_reussites() << ", "
        << "Nb. resultats (total): " << r.b_resultats() << endl;
}
```

Limites du secret

Il y a une différence entre cacher un type et cacher un nom. En C++, plusieurs mécanismes reposent sur les noms et sur leur exposition, mais ne pas avoir accès au nom d'un objet n'empêche pas nécessairement de l'utiliser.

L'exemple à droite illustre ce fait : dans `main()`, on ne peut déclarer un `X::Y` puisque la classe `Y` est interne et privée à `X`. Cependant, il est possible d'accéder à un `X::Y` et d'en utiliser les services si `X` expose ce type d'autres manières, que ce soit en utilisant `auto` à titre de type ou en n'utilisant que des instances anonymes de `X::Y` pour donner deux exemples.

```
class X {
   struct Y {
      int f() const {
         return 3;
      }
   };
public:
   Y y() const {
      return Y{};
   }
};
#include <iostream>
int main() {
   using namespace std;
   X x;
   // X::Y y0 = x.y(); // illégal, X::Y est privé
   auto y1 = x.y();
   cout << y1.f() << endl; // Ok
   cout << x.y().f() << endl; // Ok aussi
}
```

Dans d'autres langages

En Java, une classe imbriquée peut être un membre d'instance ou un membre de classe. Une classe interne appartenant à la classe englobante sera qualifiée `static` alors qu'une classe interne appartenant à chaque instance, sur une base individuelle, elle, ne le sera pas.

Dans l'exemple à droite, `main()`, une méthode de classe, ne peut instancier la classe `InterneA`, qui est un membre d'instance, mais `f()`, une méthode d'instance, le peut.

Comme tous les membres d'un objet, une classe interne peut être privée, protégée, publique ou (en Java) privée au paquetage.

```
public class Z {
   public class InterneA {
   }
   public static class InterneB {
   }
   public void f () {
      InterneA ia = new InterneA(); // ok
      InterneB ib = new InterneB(); // ok
   }
   public static void main (String [] args) {
      InterneA ia = new InterneA(); // illégal
      InterneB ib = new InterneB(); // ok
      Z z = new Z();
      z.f();
   }
}
```

En C++ et dans les langages `.NET`, une classe imbriquée est nécessairement un membre de classe. Les règles syntaxiques sont les mêmes, à ceci près que si la classe `Interne` est interne à la classe `Externe`, son nom sera `Externe::Interne` en C++ et `Externe.Interne` en C# ou en `VB.NET`.

Techniques de clonage

Le clonage est une technique importante pour les classes polymorphiques. Imaginons, pour comprendre la problématique, la situation suivante :

- il existe une hiérarchie de classes dont la racine est la classe abstraite `Image`, et dont les dérivés sont `ImageJpeg`, `ImageGif` et `ImagePng` (par exemple);
- un sous-programme reçoit en paramètre une indirection vers à une `Image`. Ce sous-programme a donc accès à l'un des dérivés de `Image`, sans savoir lequel; et
- le sous-programme doit opérer de manière à modifier l'`Image` reçue en paramètre, mais ne souhaite pas modifier l'`Image` originale. Il doit donc créer une copie de l'`Image` reçue.

Utiliser l'opérateur d'affectation sur `Image` n'est pas une option : quel serait le type de l'opérande de gauche? Le même irritant se produit avec le constructeur de copie : construire une instance de quel type?

```
void f(Image &img) {
   // Illégal: Image est abstraite
   Image image = img;
}
```

Que faire pour dupliquer correctement ce qui se cache derrière une abstraction sans savoir de quoi il s'agit?

En fait, la seule entité qui serait vraiment capable de copier une `Image` dans ce cas-ci, c'est l'instance dérivée d'`Image` vers laquelle on pointe, et elle seule.

Nous reviendrons sur certaines questions plus avancées quant au clonage dans [POOv03].

La clé est donc de demander à l'instance de produire subjectivement une copie d'elle-même; *de lui demander de se cloner*.

Comme nous le verrons dans la section ***Dans d'autres langages***, plus loin, le recours au clonage est répandu dans les langages `OO`, mais est implémenté sous plusieurs formes en fonction des coutumes et des préférences philosophiques. L'optique que nous allons proposer ici ressemble à celle que préconise Java, mais en tirant avantage des forces de C++. Le système de types de C++, contrairement à ceux de Java et des langages `.NET`, permet un clonage sans bris d'encapsulation (voir ***Covariance : spécialisation des types des méthodes polymorphiques***).

Duplication et indirections

Plusieurs langages `OO` commerciaux (dont Java et les langages `.NET`) ne supportent pour les objets qu'une sémantique d'accès indirect (voir [POOv00], ***Appendice 00***) : les objets y sont toujours créés dynamiquement avec `new` et sont manipulés de manière indirecte.

Adopter un système qui omet les types valeurs a pour effet qu'il n'est pas sécuritaire d'y dupliquer un objet à l'aide d'une construction par copie, *sauf* si l'objet se duplique lui-même, subjectivement, du fait que le code client ne sait jamais, pour une classe polymorphique `X` donnée, s'il manipule un `X` ou l'un de ses dérivés.

En effet, dans le code suivant (j'ai omis VB.NET par souci d'espace mais le problème y existe aussi et sous la même forme), chaque fois, pX1 n'est pas égal au sens du même contenu que pX0 suite à sa construction :

C++

```cpp
class X
   // ...
public:
   X(const X&);
   // ...
};
class Y : public X {
   // ...
};
int main() {
   X *pX0 = new Y;
   X *pX1 = new X{*pX});
   // ne pas oublier de
   // détruire pX0 et pX1
}
```

Java

```java
class X {
   // ...
   public X(X x) {
      // ...
   }
   // ...
}
class Y extends X {
   // ...
}
public class Test {
   public static void main
      (String [] args) {
      X pX0 = new Y();
      X pX1 = new X(pX0);
   }
}
```

C#

```csharp
class X
{
   // ...
   public X(X x)
   {
      // ...
   }
   // ...
}
class Y : X
{
   // ...
}
public class Test
{
   public static void Main
      (string [] args)
   {
      X pX0 = new Y();
      X pX1 = new X(pX0);
   }
}
```

Le problème dans chaque cas tient à ce qu'on nomme en anglais du *Slicing*. C'est-à-dire que pX1 est construit à partir d'une copie manuelle de pX0, qui semble être pris pour un X mais est en réalité un Y. Toute la partie de pX0 qui n'est pas à proprement parler une partie d'un X, donc tout ce qui y est spécifique à un Y, est tranché (d'où le nom) par la construction par copie.

En effet, le caractère indirect des manipulations dans ces exemples a la propriété qu'au bout de chaque pointeur et de chaque référence pX0, on trouve non pas un X mais quelque chose qui est au moins un X. Même avec C++, qui supporte la sémantique de valeur, on n'aura d'autre choix que d'avoir recours à des stratégies de duplication subjectives pour en arriver à dupliquer des objets manipulés indirectement.

Pour les objets manipulés à travers des indirections, la seule vraie solution au problème de la copie d'objets est une solution polymorphique, donc subjective : **le clonage**. Obtenir une copie d'un objet devient une demande à cet objet de se cloner lui-même, et cette demande nécessite un appel polymorphique.

Le code ainsi ajusté devient, en Java et en C# comme suit.

Java

```
interface Clonable {
   Object cloner();
}
class X implements Clonable {
   // ...
   public X(X x) {
      // ...
   }
   public Object cloner() {
      return new X(this);
   }
}
class Y extends X {
   public Y(Y y) {
      super (y);
      // ...
   }
   public Object cloner() {
      return new Y(this);
   }
}
public class Test {
   public static void main(String [] args) {
      X pX0 = new Y();
      X pX1 = (X) pX0.cloner();
   }
}
```

C#

```
interface IClonable
{
   object Cloner();
}
class X : IClonable
{
   // ...
   public X(X x)
   {
      // ...
   }
   public object Cloner()
   {
      return new X(this);
   }
}
class Y : X
{
   public Y(Y y) : X(y)
   {
      // ...
   }
   public object Cloner()
   {
      return new Y(this);
   }
}
public class Test
{
   public static void Main(string [] args)
   {
      X pX0 = new Y();
      X pX1 = (X) pX0.cloner();
   }
}
```

Portez attention aux conversions explicites de types dans les méthodes `main()` et `Main()`. Elles sont nécessaires à cause de l'absence de méthodes aux types covariants, dont nous discuterons sous peu.

Les clés d'un bon clonage

Revenons à C++ et à des points de design pour bien comprendre les enjeux. Il y a trois clés à la réussite d'un bon clonage. Ces clés sont :

- la présence d'un ***constructeur de copie***[86] ***protégé***[87] bien écrit dans toutes les classes sujettes à être clonées, puisque l'objet se clonant voudra faire facilement un double de lui-même;

> De manière générale, le constructeur de copie d'un type polymorphique ne devrait jamais être public puisque le code client n'est pas en mesure de l'utiliser avec sagesse.

- une méthode polymorphique bien nommée et servant précisément à la tâche de clonage (souvent, on nommera cette méthode **`cloner()`**); et
- puisqu'on parle de polymorphisme, une racine bien choisie pour implanter la mécanique.

Nous examinerons d'abord le code requis pour réaliser le clonage avec la majorité des langages `OO`, puis nous examinerons plus en détail une force du système de types de C++ qui permet, avec ce langage, d'éliminer essentiellement les bris d'encapsulation qui sont une tare des mécanismes de clonage dans plusieurs autres langages.

Le choix de la racine

On a le réflexe d'y aller d'une interface globale, souvent nommée `Clonable` ou quelque chose de semblable, lorsqu'on vise à implanter une mécanique de clonage dans une hiérarchie d'objets. C'est d'ailleurs ce que font Java et les langages `.NET` dans ce genre de situation.

Ce qui est agaçant avec cette démarche, c'est que le type retourné par la méthode `cloner()` doit alors être le type le plus abstrait possible pour la mécanique de clonage. Ici, la méthode `Clonable::cloner()` retournera sans doute un `Clonable*`.

Dans les langages à héritage simple et à racine globale unique, on retournera une référence vers le seul type garanti, qui sera celui de la racine globale. Ensuite, on vérifiera le type obtenu avec les mécanismes d'inférence dynamique de types, qui sont `instanceof` en Java et `is` en C#.

Tel que discuté dans la section sur l'opérateur `dynamic_cast`, il s'agit là d'un bête bris d'encapsulation, mais auquel nous sommes contraints par la mécanique du langage.

---

[86] Évidemment, on ne parle pas d'un constructeur de copie conforme à ceux de C++ puisque de tels constructeurs impliquent des types valeurs. On parle d'un constructeur prenant en paramètre une référence sur une instance du même type que le type construit.

[87] Notez bien la qualification de protection `protected`. Ce constructeur doit être accessible aux enfants mais jamais au code client; il n'a d'utilité que celle de soutien interne à la bonne mécanique de copie. Le code client n'est pas en mesure, de manière générale, de déterminer lui-même les types réellement impliqués lorsque les types apparemment impliqués sont polymorphiques.

Une implémentation C++ complète et semblable à celle qu'on pourrait retrouver en Java ou dans un langage .NET serait :

- de créer l'interface Clonable;
- d'y déclarer la méthode abstraite cloner() retournant un Clonable*;
- de dériver Image de Clonable;
- de dériver ImageJpeg d'Image;
- d'y implémenter cloner(); et
- d'utiliser dynamic_cast sur le pointeur de Clonable retourné pour vérifier qu'il s'agit bel et bien au moins d'un Image*.

L'exemple à droite donne l'essentiel de chaque étape, en n'examinant que la spécialisation ImageJpeg (les autres spécialisations possibles sont, pour nos fins, identiques).

Remarquez au passage l'importance d'avoir un destructeur virtuel dans la classe servant de racine pour la mécanique de clonage. L'utilisateur de la mécanique de clonage est légitimé de détruire un pointeur sur cette racine, ce qui nous oblige à implanter la destruction polymorphique au moins à partir de cette classe.

```
struct Clonable {
   virtual Clonable *cloner() const = 0;
   virtual ~Clonable() = default;
};
class Image : public Clonable {
   // ...
};
class ImageJpeg : public Image {
protected:
   ImageJpeg(const ImageJpeg &);
public:
   Clonable *cloner() const override {
      return new ImageJpeg{*this};
   }
   // ...
};
void f(Image *x) {
   Clonable *temp = x->cloner();
   if (Image *p= dynamic_cast<Image*>(temp),
       p) {
      // utiliser p
   }
   delete temp;
}
```

Ici, le dynamic_cast est nécessaire pour l'obtention de code robuste, dû au passage par une abstraction de trop haut niveau (Clonable). Une question se pose cependant : veut-on réellement que cloner() retourne un Clonable*, quitte à le convertir par la suite? Rien de moins sûr.

Une solution évitant les conversions explicites de types avec dynamic_cast serait d'utiliser une méthode cloner() pour chaque classe racine qui, comme Image, est susceptible de servir comme racine pour fins de clonage.

Mais cela mène à un autre problème au moins aussi désagréable, qui est la multiplication des implantations disjointes de cloner() dans les hiérarchies de classe. On fait donc face ici à un irritant *conceptuel*, où chacune des solutions évidentes pose un problème de fond.

Covariance : spécialisation des types des méthodes polymorphiques

Une solution élégante à ce problème doit passer par un raffinement conceptuel du langage. Ainsi, en C++, il est possible pour une classe dérivée de raffiner le type d'une méthode virtuelle déclarée au moins chez son parent, dans la mesure où le type retourné par la méthode du dérivé est une spécialisation du type retourné par celle du parent.

Le type de la méthode est alors qualifié de **covariant** (voir ***Annexe 02 – Covariance et contravariance (bref)*** pour plus de détails) puisqu'il évolue de concert avec le raffinement du type de l'objet auquel il appartient.

Dans notre exemple, on pourrait conserver l'interface `Clonable` avec sa méthode abstraite `cloner()` retournant un `Clonable*`, mais on pourrait spécialiser dans `Image` la méthode `cloner()` comme retournant un `Image*`, et faire de même pour `ImageJpeg`, dans la mesure où `Image` a pour ancêtre `Clonable`, et dans la mesure où le parent de `Image` retournait, par sa méthode `cloner()`, un pointeur vers un ancêtre de `Image`.

Verbaliser cette stratégie est un peu lourd, mais le concept est, au fond, simple :

- de créer l'interface `Clonable`;
- d'y déclarer la méthode abstraite `cloner()` retournant un `Clonable*`;
- de dériver `Image` de `Clonable`;
- de dériver `ImageJpeg` d'`Image`;
- d'y implémenter `cloner()` de manière covariante (voir le code à droite).

Ceci donne un résultat sans danger :

- si quelqu'un appelle la méthode `cloner()` d'un `Image*` à travers un `Image*`, ceci lui donnera un `Image*`, ce qui est au moins un `Clonable*` (et même plus!). Rien n'est perdu en comparaison avec la version retournant simplement un `Clonable*`;
- par contre, à travers un `Clonable*`, un appel à `cloner()` retournera un `Clonable*`, ce qui était déjà le cas auparavant. De toute manière, un `Image*` *est* un `Clonable*`.

```
struct Clonable {
   virtual Clonable *cloner() const = 0;
   virtual ~Clonable() = default;
};
class Image : public Clonable {
protected:
   Image(const Image&);
public:
   Image *cloner() const = 0;
   // ...
};
class ImageJpeg : public Image {
protected:
   ImageJpeg(const ImageJpeg&);
public:
   Image *cloner() const {
      return new ImageJpeg{*this};
   }
   // ...
};
void f(Image *x) {
   Image * img= x->cloner();
   // utiliser img
   delete img;
}
```

Ceci rend le `dynamic_cast` superflu pour la plupart des cas envisageables.

À titre d'exercice, notez que `f()` ci-dessus est à risque si une exception est levée. Comment pourrions-nous la sécuriser?

Copier ou cloner?

La question de la duplication d'un objet, qu'il soit accédé directement ou non, est une question complexe se déclinant selon les angles suivants :

- la copie de contenu devrait-elle être possible sur un type donné? Si oui, dans quels cas?
- pour un type donné, la copie de contenu devrait-elle être polymorphique (donc subjective) ou automatique et implicite (donc objective)?
- si un objet est partageable, devrait-il être *incopiable*?

Ce qui suit présume que vous avez lu et compris l'***appendice*** **`00`** de [POOv00] portant sur les sémantiques d'accès directe et indirecte aux objets.

Distinguer copie et clonage

Commençons par distinguer les concepts de *copie* et de *clonage*. Pour le code client :

- la **copie** duplique le contenu d'un objet en présumant de son type effectif. Le constructeur par copie et l'opérateur d'affectation sont des exemples classiques d'opérations de copie. Les types valeur et les classes concrètes prennent en partie leur sens de la présence d'une mécanique de copie correcte et efficace;
- le **clonage** demande à un objet de produire une copie de lui-même. Une telle opération prend son sens lorsque le type effectif de l'objet à dupliquer n'est véritablement connu que de l'objet lui-même, ce qui implique que les méthodes de clonage sont habituellement polymorphiques et utilisées à travers des abstractions.

Dans les langages supportant une sémantique directe d'accès aux objets, la copie joue un rôle prédominant. Permettre à un objet de contrôler les moments charnières de son existence telles que la construction, la destruction et la copie a pour effet de faciliter la production de code robuste et performant à tous points de vue.

Dans les langages se limitant à une sémantique indirecte d'accès aux objets, la copie joue un rôle beaucoup moins important. L'affectation se limite habituellement à une copie de référence et le passage de paramètres par valeur copie des références plutôt que des objets.

Stratégie de duplication et nature des objets

La question, posée simplement, est de déterminer comment les instances d'une classe donnée devraient être dupliquées. La réponse, de manière peut-être surprenante, est relativement simple et peut pratiquement s'exprimer sous forme de recette.

Quand un objet a un comportement polymorphique, donc lorsqu'un objet expose au moins une méthode virtuelle ou lorsque cet objet a un parent *clonable*, il est probable que le clonage soit le choix à faire. Normalement, un tel objet devrait avoir un ***constructeur de copie protégé***, de ***bloquer l'affectation*** et d'exposer une ***méthode polymorphique de clonage***.

Si au moins une méthode est polymorphique, alors le constructeur de copie devrait être protégé et l'objet devrait offrir une méthode de clonage publique.

Quand un objet est un type valeur, donc quand il n'expose aucune méthode polymorphique, alors une méthode de clonage est superflue (elle grossit l'objet sans apporter de fonctionnalité utile) et devrait être évitée. L'approche à privilégier pour ces objets est plutôt d'exposer un ***constructeur de copie public*** et un ***opérateur d'affectation public***. Dans bien des cas, laisser s'appliquer la Sainte-Trinité suffira.

Si aucune méthode n'est polymorphique, alors il ne devrait pas y avoir de méthode de clonage et le constructeur de copie devrait être public.

La raison pour laquelle un constructeur par copie devrait être implémenté mais protégé même dans un type *clonable* est que le constructeur par copie est la meilleure manière de réaliser, en fin de compte, une copie d'objet (même dans les langages à sémantique indirecte seulement comme java et les langages `.NET`).

Le clonage est une stratégie polymorphique pour réaliser une copie d'une manière subjective. Une méthode polymorphique est invoquée sur un objet en tant que tel et la mécanique du polymorphisme sert à trouver *la bonne méthode* pour *le bon objet*. Lorsque la méthode effective de clonage a été invoquée (lorsque le type d'objet à dupliquer a été repéré), dans la méthode implémentée elle-même, l'objet à dupliquer peut se copier lui-même à l'aide d'une construction par copie puisque le type d'objet à copier devient connu, sans ambigüité.

Remarques supplémentaires sur la duplication d'objets

Cette petite section sur la copie et le clonage polarise des éléments qui se retrouvent à plusieurs endroits dans ces notes de cours.

Certains types doivent être *incopiables*. Vous trouverez des explications en ce sens à la fois plus haut, section ***Hériter pour contraindre***, et dans [POOv02], section ***Singletons***.

Le cas des singletons est probant puisque ce schéma de conception, par définition, implique la mise en place d'une stratégie faisant en sorte qu'il n'y ait qu'une seule instance de la classe qualifiée de singleton dans un programme tout entier. Permettre la copie d'objets dans une telle situation serait un non-sens.

En situation de partage, la question de la duplication devient complexe. Certains programmes sont faits de multiples fils d'exécution concurrents (de multiples *threads*) sujets à partager un même objet. Dans ces programmes, la question de savoir qui sera responsable de détruire l'objet devient complexe, parfois même indéterministe.

La solution à ces problèmes se trouve habituellement parmi les suivantes :

- dupliquer systématiquement les objets. Ceci règle le problème de la responsabilité quant à la destruction de l'objet (chaque client de l'objet doit détruire sa propre copie) mais empêche, en retour, le véritable partage de l'objet;
- partager les objets en comptabilisant dans chaque objet le nombre de ses clients. Ceci responsabilise l'objet quant à sa propre destruction (il se fait hara-kiri lors de la déconnexion de son dernier client) mais implique une comptabilité manuelle pouvant devenir complexe. La section ***Pointeurs intelligents et objets partageables*** de [POOv02] aborde ce sujet mais une implémentation solide est un sujet plus avancé;
- responsabiliser le moteur du langage pour la prise en charge de la durée de vie des objets, ce qui constitue l'approche choisie par les langages munis d'un ramasse-miettes comme Java et les langages `.NET`. Cela a en retour le gros défaut de priver les objets de la capacité d'être pleinement encapsulés du fait que leur véritable libération devient, en général, indéterministe.

La situation est encore plus complexe dans les systèmes répartis ou incluant des éléments de plusieurs langages. Il n'y a pas de solution universelle au problème de la gestion des objets partagés, chaque approche ayant ses avantages et ses inconvénients.

Notez que la sémantique indirecte, qui mène intrinsèquement à privilégier le clonage à la copie, entraîne aussi souvent, dans les langages n'offrant que la sémantique d'accès indirect aux objets, une obligation[88] de programmation défensive, chose très coûteuse en pratique. Aucune solution n'est parfaite; le choix d'une approche, quelle qu'elle soit, est chose politique.

[88] Du moins dans les produits qui se veulent sécuritaires.

Dans d'autres langages

Le clonage est un thème récurrent dans les langages OO. Java et les langages .NET ont, à même leur abstraction fondamentale (Object en Java, object en C#), une méthode polymorphique permettant de cloner tout instance d'une classe dérivée de cette classe racine, donc tout objet du langage.

Le clonage étant nécessaire en Java et dans les langages .NET, il existe déjà une stratégie de clonage formelle sur ces deux plateformes :

- la classe object des langages .NET expose une méthode protégée nommée MemberwiseClone(). Cette méthode retourne une référence vers un object, donc vers le clone créé. Cette méthode ne peut être surchargée, et crée une copie de surface (*Shallow Copy*) de l'objet cloné. Cela signifie que si l'objet cloné contient des références vers d'autres objets, alors ceux-ci ne sont pas clonés au passage, donc le clone et le cloné réfèrent tous deux aux mêmes objets;
- il existe une autre approche pour les langages .NET, mais elle devrait être évitée. La première livraison de l'interface ICloneable de cette plateforme a souffert d'une documentation incomplète et a mené à des implémentations divergentes, oscillant entre une copie de surface et une copie en profondeur. Les architectes de l'infrastructure .NET recommandent de l'éviter [FwkDes];
- Java expose une méthode protégée clone() à même Object. Cette méthode retourne une référence à un Object, donc vers le clone créé. Elle doit être surchargée pour permettre le clonage de manière standardisée, et a pour rôle de créer un double de l'objet cloné. Ce clone peut être une copie de surface ou une copie en profondeur (*Deep Copy*), selon la sémantique de la classe;
- en Java, notez que les tableaux implémentent une méthode clone() qui fait une copie de contenu élément par élément, sans toutefois cloner les éléments (ce qui résulte en une copie de surface).

Les systèmes de types à racine unique, comme ceux de Java et des langages .NET, imposent des passages par l'abstraction principale de la plateforme (Object, typiquement) et entraînent une prolifération des conversions explicites de types.

Java supporte les types de retour covariants.

Nécessité et habitude

Il importe de noter que plusieurs programmeurs Java ou .NET ne sont pas conscients du clonage, ou même du fait qu'il est important de porter attention à ces détails. Ceci tient à plusieurs raisons :

- le clonage n'est vraiment utile que si une méthode compte modifier un objet mais est susceptible de vouloir conserver l'original intact. C'est un cas d'utilisation commun, soit, mais sans être le plus fréquemment rencontré d'entre tous;
- pour tout ce qui touche à la multiprogrammation, il importe de s'assurer d'éviter que deux *threads* modifient concurremment un même objet. Deux options existent dans ces langages : le clonage, bien sûr, mais aussi les objets immuables ([POOv00], ***Copies et classes d'objets immuables***);
- ces deux cas d'utilisation sont plus fréquents dans des applications complexes ou de taille importante que dans des applications plus « éducatives »;
- connaissant les risques associés au partage d'un objet, les conceptrices et les concepteurs de Java et des langages .NET ont fait en sorte que plusieurs classes clés de leurs bibliothèques respectives (pour Java : `String`, `Integer`, `Boolean`, etc.) soient immuables. Une classe dont les instances sont immuables n'a pas besoin de clonage, par définition.

Il demeure que comprendre le clonage accroît l'étendue de vos compétences, de même que la richesse de votre boîte à outils, donc le nombre de catégories d'applications que vous serez en mesure de rédiger.

Exploiter l'idiome RAII[89]

Le langage C++ possède un avantage considérable sur plusieurs autres langages OO en ce sens qu'il offre un support complet à l'encapsulation à l'aide de destructeurs déterministes[90].

Ceci signifie que dans un sous-programme comme f() proposé à droite, l'instance x de la classe X sera nécessairement détruite, peu importe la manière ou le lieu où se complétera f() : une exception locale ou dans un sous-programme appelé par f(), un return impromptu, la rencontre de l'accolade fermante de la définition de f(), etc.

```
void f() {
   X x;
   // ...
}
```

L'invocation du destructeur d'un objet alloué automatiquement est déterministe au sens où il sera invoqué de manière prévisible dès la fin de sa portée. Ceci a mené au développement d'idiomes de programmation sécuritaires et très utiles, en particulier l'idiome RAII (pour *Resource Acquisition Is Initialization*) qui fait partie des techniques les plus utiles et les plus simples ayant été développées par les gourous du langage.

L'idée de base est simple : le constructeur et le destructeur d'un objet étant des opérations symétriques, et le destructeur d'un objet local à une portée étant détruit à la fin de cette portée peu importe l'événement y ayant mené[91], il est possible d'automatiser la libération de ressources à l'aide d'un destructeur dans la mesure où la ressource est connue de l'objet à détruire… et le constructeur n'est-il pas un bel endroit pour saisir une ressource?

Par exemple, examinons le code de f() proposé à droite. En surface, mais en surface seulement, ce code semble sécuritaire. Cette procédure :

- valide n avant de créer un tableau de X;
- instancie ce tableau, qui contiendra n instances « typiques » de X;
- sollicite g() en lui fournissant à la fois le tableau et sa taille; et enfin
- détruit le tableau.

```
// ...
class TailleIllegale {};
void f(int n) {
   if (n <= 0)
      throw TailleIllegale{};
   X *p = new X[n];
   g(p, n);
   delete [] p;
}
// ...
```

Réflexion 01.1 : pourquoi la fonction f() n'annonce-t-elle pas throw(TailleIllegale), et ne l'annoncerait pas même si les spécifications d'exceptions n'étaient pas dépréciées (comme elles le sont depuis C++ 11)? Cela mérite réflexion, alors essayez d'y répondre avant de tourner la page; voir ***Réflexion 01.1 : annoncer les exceptions*** pour des détails.

---

[89] L'idiome RAII a été décrit et utilisé à quelques reprises dans [POOv00].

[90] Les langages à objets accédés indirectement et à collecte automatique d'ordures ramassent aussi les objets qu'ils prennent en charge mais ont beaucoup plus de difficulté à gérer harmonieusement les ressources externes, ce qui explique que le code écrit dans ces langages et intégrant des ressources externes foisonne d'opérations manuelles de libération de ressources et ait besoin de blocs finally dans son traitement d'exceptions. Heureusement, C# supporte maintenant les blocs using, et Java implémente les blocs *Try-With* depuis la version 7 du langage.

[91] Levée d'exception, sortie brusque résultant d'un appel à exit(), return en milieu ou en fin de sous-programme… Tout ce qui se gère dans le flot des opérations d'un programme, en fait.

Malheureusement, le portrait de `f()` est moins rose qu'il n'y paraît à première vue :

- le constructeur par défaut de `X` peut, à l'occasion, lever une exception;
- de même, `g()` peut lever une exception; or
- si une exception est levée avant l'appel à `delete [] p`, alors cette opération ne sera jamais exécutée.

Heureusement, le premier cas (que `X::X()` lève une exception) ne nous préoccupe pas vraiment puisque le standard de C++ nous dit que si une exception est levée dans le constructeur d'un objet dans un tableau, alors les éléments déjà construits du tableau seront détruits en ordre inverse de celui de leur construction.

Le deuxième cas, par contre, est un irritant majeur car, dans la procédure `f()` telle que présentée, cela impliquerait une fuite de mémoire et cette fuite serait notre responsabilité. En effet, `f()` n'invoquant pas `g()` dans un bloc `try`, il ne lui sera pas possible de se saisir d'une exception et de libérer le tableau. Si `g()` lève quelque exception que ce soit, un tableau d'au moins `n*sizeof(X)` *bytes* ne sera pas libéré et, ce qui peut être pire encore, le destructeur de `nbElems` instances de `X` ne sera jamais invoqué.

> Le fait que les destructeurs des `n` instances de `X` ne soient pas invoqués est en général pire que le fait que le tableau lui-même ne soit pas libéré, du fait qu'il est possible de collecter le tableau flottant avec une collecte automatique d'ordures comme celle proposée par C++ `11` mais que la non-invocation du code des destructeurs, elle, de laisser ouvertes des ressources arbitrairement importantes.

Une solution sans `RAII`, semblable à ce qu'on aurait en Java ou dans un langages `.NET` si le problème se portait de la gestion d'un `X` à celle d'une ressource externe comme une connexion à une base de données ou à un *socket*, serait celle proposée à droite si `g()` était susceptible de lever quelque exception que ce soit.

```
// ...
class TailleIllegale {};
void f(int n) {
   if (n <= 0)
      throw TailleIllegale{};
   X *p = new X[n];
   try {
      g(p, n);
      delete[] p;
   } catch (...) {
      delete[] p;
      throw;
   }
}
// ...
```

Cette solution est à la fois laide et complexe, nécessitant de la prudence et de l'artisanat pour couvrir tous les cas possibles de sortie impromptue et de libération manuelle de ressources. Ici, le code appelant, `f()`, doit prévoir les incidents à l'invocation de `g()`, libérer `p` que `g()` se soit exécuté normalement ou qu'elle ait levé une exception, et (ne sachant pas ce qui a été levé) doit faire *catch-any* un *re-throw* dans le bloc `catch`.

> Heureusement, dans les langages où de telles structures de contrôle constituent les seules avenues possibles, on aura typiquement un bloc `finally` (voir [POOv00]) pour gérer la libération des ressources et éviter de les placer à la fois dans le bloc `try` et dans le bloc `catch`.

Cette façon de faire est aussi difficile à échelonner à des problèmes plus complexes du fait que le nombre de cas particuliers tend à exploser rapidement.

Personne, programmeuse/ programmeur comme gestionnaire, ne souhaite gérer du code de ce genre.

L'idiome `RAII` est une solution simple, élégante et d'une grande efficacité pour ce genre de situation. Voici comment la mettre en place :

- l'idiome `RAII` utilise un objet, ici une instance de la classe (au nom un peu verbeux) `AutoTableau`;
- le constructeur de cet objet prend en paramètre un pointeur. Ce faisant, *il est initialisé par acquisition d'une ressource*;
- on considère que l'instance `at` d'`AutoTableau` est responsable, à partir de ce point, de la libération éventuelle de ce pointeur;
- le destructeur de `at` détruira le tableau se trouvant au bout du pointeur reçu lors de sa construction;
- pour utiliser correctement un `AutoTableau`, la procédure `f()` l'nstancie tout simplement au moment opportun et lui passe en paramètre le tableau qui devra être libéré… et c'est tout!

Remarquez que `f()` ne s'occupe plus ni du traitement d'exceptions levées par `g()`, ni de détruire manuellement le tableau créé dynamiquement dans `f()`. En effet :

- si `f()` se termine normalement, alors l'instance `at` sera détruite et son destructeur libérera le tableau; et
- si `g()` lève une exception, alors les destructeurs des objets locaux à `f()`, incluant `at`, seront appelés;
- toute exception levée par `g()` se propagera normalement vers l'appelant de `f()`.

```
// ...
// Incopiable : voir plus haut
class AutoTableau : Incopiable {
   X *p_;
public:
   AutoTableau(X *p) noexcept : p_{p} {
   }
   X* get() noexcept {
      return p_;
   }
   const X* get() const noexcept {
      return p_;
   }
   ~AutoTableau() {
      delete[] p_;
   }
};
class TailleIllegale {};
void f(int n) {
   if (n <= 0)
      throw TailleIllegale{};
   AutoTableau at{new X[n]};
   g(at.get(), n);
}
// ...
```

Notez qu'il est important de nommer l'objet RAII. Si nous avions négligé de donner un nom à l'objet at, déclarant par exemple

```
AutoTableau {new X[n]};
```

plutôt que

```
AutoTableau at {new X[n]};
```

alors une instance d'AutoTableau aurait été créée, puis détruite dans la même expression, entraînant avec elle le tableau.

Le code de f() est nettement allégé, simplifié, clarifié. En injectant une stratégie OO dans le portrait, nous venons d'accroître grandement la qualité globale de notre code.

Le cas de la copie

Un objet implémentant l'idiome RAII tend à être destiné à une durée de vie restreinte, souvent locale à une fonction, mais certains peuvent vivre plus longtemps. Sans entrer dans les détails, notez que la duplication d'un objet RAII est toujours subtile parce qu'elle pose la question de la responsabilité à l'égard de la ressource encapsulée. Une solution simple est de ne jamais passer un objet RAII par copie à un sous-programme, de privilégier par exemple les indirections, les objets incopiables ou les objets partageables (voir [POOv02], ***Pointeurs intelligents et objets partageables***).

Solution plus générale

La stratégie RAII exposée par AutoTableau est très pointue, acquérant puis libérant un tableau de X. On serait en droit de se demander s'il est raisonnable de concevoir une classe pour chaque type d'objet à libérer automatiquement.

Heureusement, bien qu'il soit possible de créer des classes au cas par cas, ce n'est ni souhaitable, ni nécessaire. La bibliothèque standard [POOv02] expose entre autres la classe générique std::unique_ptr pour couvrir des cas de pointeurs autres que les pointeurs sur des tableaux, donc nous pouvons y avoir recours (unique_ptr remplace *très* avantageusement auto_ptr qui faisait le travail mais avait des défauts très déplaisants).

De même, des stratégies de programmation générique [POOv02] permettent de couvrir tous les cas en bloc. Voici comment, à titre d'avant-goût, la programmation générique et la POO peuvent, lorsque prises ensembles, rendre notre vie nettement plus agréable et notre code nettement supérieur.

On peut transformer la classe AutoTableau pour qu'elle passe d'une classe gérant la libération automatique d'un tableau de X à une classe gérant la libération automatique d'un tableau de T pour un type T quelconque[92], par l'injection de la mention template <class T> avant le nom de la classe et par le remplacement de X par T dans le corps de la classe.

Ce faisant, lorsque le compilateur constate l'utilisation d'un AutoTableau appliqué à un type donné (au type X dans le cas de la procédure f()), il génère le code requis à notre place.

Le code aurait pu être écrit manuellement, mais il est nettement préférable de laisser le compilateur générer le nécessaire à notre place, n'est-ce pas?

L'idiome RAII sert à une multitude de fins, en fait à tout ce qui nécessite une forme de symétrie (ouvrir/ fermer un fichier; ouvrir/ fermer une connexion à un *socket*; charger/ décharger une bibliothèque à liens dynamiques; *etc*.). Il augmente à la fois la qualité et la sécurité du code.

```
// ...
template<class T>
   class AutoTableau : Incopiable {
      T *p_;
   public:
      AutoTableau(T *p) noexcept : p_{p} {
      }
      T* get() noexcept {
         return p_;
      }
      const T* get() const noexcept {
         return p_;
      }
      ~AutoTableau(){
         delete [] p_;
       }
   };
class TailleIllegale {};
void f(const int n) {
   if (n <= 0)
      throw TailleIllegale{};
   AutoTableau<X> at{new X[n]};
   g(at.get(), n);
}
// ...
```

> Depuis C++ 11, il existe une solution officielle à ce problème : la classe unique_ptr. Voir [POOv02], ***Pointeurs intelligents et objets partageables*** pour en savoir plus.

---

[92] Oui, cela fonctionne pour tout type, qu'il s'agisse d'une classe, d'un struct ou d'un type primitif.

Dans d'autres langages

En Java et en `VB.NET`, l'idiome `RAII` n'est pas applicable. Le code client est responsable de la libération des ressources externes, et la finalisation est non déterministe. Les blocs `finally` sont tributaires de cette réalité.

En C#, un objet manipulant des ressources externes et exposant une interface spéciale du monde `.NET` (l'interface `IDisposable`) peut être placé par le code client dans un bloc `using`.

Le compilateur insère alors silencieusement des blocs `try`, `catch` et `finally` autour de l'utilisation de l'objet visé (à droite, l'objet `x`) et s'assure d'invoquer correctement la méthode `Dispose()` de son interface `IDisposable` à la fin du bloc, que cette fin soit atteinte normalement ou dû à une exception.

```
using (X x = new X())
{
   // utilisation de x
}
```

Les blocs `using` de C# constituent du sucre syntaxique allégeant la tâche du code client, mais laissent tout de même la responsabilité de la libération des ressources sur les épaules de ce dernier, résultat du non déterminisme de la finalisation.

Depuis Java `7`, il est possible avec ce langage de manipuler des instances de classes implémentant `AutoCloseable` et sa méthode `close()` en utilisant un bloc *try-with*.

Là où C# exige une méthode `Dispose()`, Java exige une méthode `close()`. Pour le reste, la mécanique est la même, soit l'insertion d'un bloc `finally` implicite au choix du code client.

```
try (X x = new X()) {
   // utilisation de x
}
```

Enfin, brève parenthèse : la généricité (aperçue brièvement avec la mention d'un `template` plus haut) est possible en Java, C# et `VB.NET`, mais pour une généricité seulement sur les classes, pas sur les types primitifs. Le document [POOv02] discute longuement de généricité, un sujet de la plus haute importance.

Implanter un traitement d'exceptions rigoureux en C++

Quelques remarques sur des stratégies élégantes pour faciliter l'implantation d'un traitement d'exceptions propre et rigoureux en C++.

Rédiger du code offrant un traitement d'exceptions rigoureux est à la fois nécessaire et difficile, dans tous les langages `OO` mais pour des raisons variant en fonction de la philosophie sous-jacente au langage choisi :

- les langages offrant un destructeur déterministe comme C++ doivent faire avec la construction et la destruction implicite de variables temporaires (car retourner un objet par valeur peut impliquer la levée d'une exception dans son constructeur!) mais peuvent automatiser les opérations de nettoyage lorsqu'une exception se produit dans un sous-programme grâce aux destructeurs des objets qui y sont déclarés localement; alors que
- les langages à construction explicite et à collecte automatique d'ordures (Java, C#, `VB.NET`) requièrent du code de nettoyage explicite (bloc `finally` et opérations manuelles de libération de ressources externes) mais vivent moins de risques d'effets de bord involontaires du fait que la construction des objets y est toujours faite explicitement (par un appel à `new`).

Facteurs de rigueur : stabilité et neutralité

On exige d'un programme se voulant robuste deux facteurs de rigueur dans la gestion des cas d'exception, soit que ses objets soient ***stables*** lors de la levée d'une exception (*Exception-Safe*) et qu'ils soient ***neutres*** face aux exceptions (*Exception-Neutral*).

La **stabilité** exige d'un objet levant une exception au cours de l'exécution d'une méthode :

- qu'il ne plante pas, sauf s'il serait préférable de le faire;
- que ses actions n'entraînent pas de fuite de ressources; et
- que son état soit stable, donc qu'il ne perde pas d'information et ne tombe pas dans un étant incohérent.

Le cas archétypal[93] est celui d'une pile dont on voudrait obtenir un élément, par définition celui sur le dessus.

Un code *Exception-Safe* sera tel qu'il sera démontrable que jamais cet objet ne perdra accidentellement l'élément sur le dessus, et que jamais la vision qu'il a de lui-même, de ses propres états, ne deviendra incohérente.

```
class Element { /* ... */ };
class Pile {
   enum { CAPACITY = 1'000 };
   Element elems[CAPACITY];
   int cur; // point d'insertion
public:
   // ...
   Element pop() {
      --cur;
      return elems[cur];
   }
   // ...
};
```

L'exemple ci-dessus, par exemple, ne sera considéré stable que s'il est possible de démontrer que le constructeur de copie d'un `Element` ne lèvera jamais d'exception. Dans le cas contraire, l'élément sur le dessus de la pile pourrait être perdu sans jamais avoir pu être récupéré.

[93] Voir à cet effet [UnsStack] qui a lancé (à l'époque) de chauds débats dans la communauté Internet.

Comparez la version plus haut avec celle visible à droite, qui est stable mais offre une interface différente.

La méthode `top()` ne lèvera pas d'exception (outre peut-être `Pile::Vide`) du fait qu'elle retourne une référence sur l'élément situé sur le dessus de la pile. Ne générant pas de copie, elle n'est pas sujette aux particularités de la classe `Element` (retourner un type primitif comme une référence ne lèvera jamais d'exception en C++). Une version `const` de la méthode `top()` aurait aussi pu être offerte.

La méthode `pop()` ne lèvera pas non plus d'exception (outre `Pile::Vide` encore une fois) du fait qu'elle ne fait que décrémenter un compteur entier.

Rédiger la classe `Pile` de manière à la rendre (au moins en partie) indépendante des détails de la classe `Element` est possible, mais demande une conception prudente de son interface.

```
class Element { /* ... */ };
class Pile {
   enum { CAPACITY = 1'000 };
   Element elems[CAPACITY];
   int cur; // point d'insertion
public:
   // ...
   class Vide {};
   bool empty() const noexcept {
      return cur == 0;
   }
   Element& top() {
      if (empty()) throw Vide{};
      return elems[cur-1];
   }
   void pop() {
      if (empty()) throw Vide{};
      --cur;
   }
   // ...
};
```

Le cas de la pile est un cas subtil, détaillé dans [ExcCpp], qui explique entre autres pourquoi la bibliothèque standard de C++ offre une classe `std::stack` (une pile) dont la méthode `pop()` est de type `void` et qui force le code utilisateur à utiliser une autre méthode (`top()`) pour consulter l'élément sur le dessus de la pile en question: cela permet à cette classe d'être stable. La rigueur est, on le comprendra, particulièrement importante dans du code de bibliothèque!

C++ est un langage `OO` qui supporte en plus la programmation générique à travers les *templates* [POOv02], qui sont des classes ou des sous-programmes opérant sur des types pris en un sens générique. Il est parfois (mais pas toujours) possible d'écrire du code à la fois stable et neutre quant aux exceptions en opérant sur des types dont on ne connaît pas la nature.

La **neutralité** implique qu'un objet rencontrant une levée d'exception dans l'exercice de ses fonctions laisse transparaître cette exception (ou une exception cohérente) au code client, plutôt que de la consommer lui-même et de la remplacer par autre chose. L'idée maîtresse derrière le traitement d'exceptions, après tout, est de reconnaître que les tâches de reconnaître un problème et de gérer un problème sont foncièrement différentes et demandent toutes deux un savoir et des aptitudes différentes.

Pour une discussion plus en détail à propos de ces quelques sujets, voir [BooExcS].

Deux exemples de non neutralité face aux exceptions sont proposés à droite avec les fonctions `ga()` et `gb()` :

- la fonction `ga()` consomme l'exception levée par `fb()`, s'il y a lieu, et retourne une valeur par défaut dans le cas où une telle exception est levée. Ce faisant, `ga()` dissimule la levée d'exception au yeux du code client, ce qui complique la mise au point des programmes;
- la fonction `gb()` capte une exception d'un type et la remplace, aux yeux du code client, par une exception d'un autre type.

Un exemple de sous-programme neutre est la fonction `g()`, qui réalise plusieurs opérations susceptibles de lever des exceptions mais gère des ressources localement et doit les nettoyer avant de laisser filtrer les exceptions au code client.

Cette fonction attrape ce qui est levé (s'il y a lieu) avec un `catch(...)` et relance ce qui fut attrapé, quoi que ce soit, une fois le nettoyage réalisé.

La neutralité n'est pas un dogme, mais bien une saine pratique lorsque prise dans un sens général. Il existe en effet des cas où la neutralité n'est pas indiquée.

Le cas classique est celui des destructeurs, qui ne devraient jamais laisser filtrer d'exceptions : si un destructeur doit par exemple fermer une connexion à une base de données et si l'opération de fermeture peut lever une exception, mieux vaut faire en sorte que le destructeur consomme silencieusement l'exception et produise un rapport d'erreur sur un média de masse que laisser filtrer l'exception et courir le risque de détruire complètement la stabilité du programme.

```
int fa();
int fb(); // throw(Z);
int ga() {
   try {
      return fb();
   } catch(Z) {
      return 0; // pas neutre
   }
}
int gb() { // throw(X)
   try {
      return fb();
   } catch(Z) {
      throw X{}; // pas neutre
   }
}
int g() { // throw(X,Y,Z)
   int *p = nullptr;
   try {
      int nbElems = fa();
      p = new int[nbElems];
      int val = fb();
      // ...
      delete[] p;
   } catch (...) { // X, Y ou Z
      // nettoyage
      delete[] p;
      throw; // neutre
   }
}
```

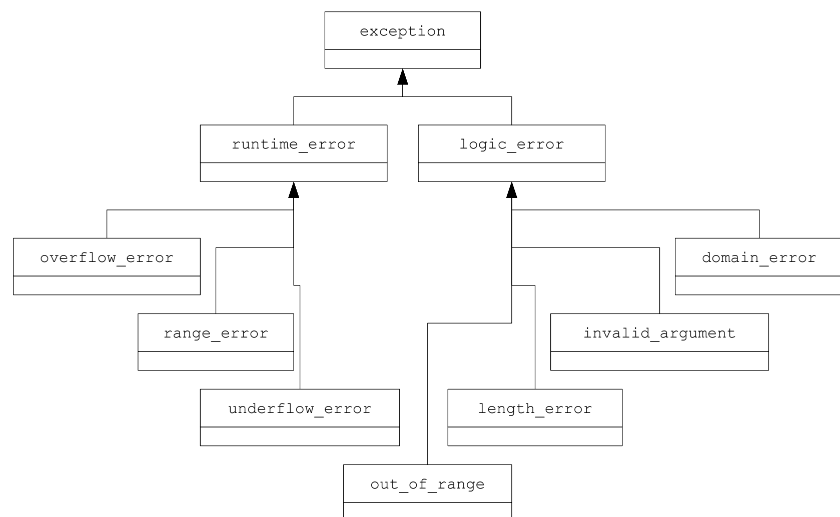
Types standards d'exceptions

C++ offre un petit nombre de classes standards d'exceptions, déclarées dans `<stdexcept>` et faisant partie de l'espace nommé `std`. Si le problème que rencontre votre programme cadre bien avec l'une ou l'autre de ces classes, il peut être sage de lever une exception d'un type standard plutôt qu'une exception d'un type maison.

Ceux de C++ 03 suivent; C++ 11 offre une gamme plus riche, détaillée dans [CppStdLib].

Type	Sémantique attendue
`logic_error`	Devrait servir de parent aux exceptions représentant des erreurs de logique, souvent susceptibles d'être détectées avant l'exécution du programme.
`domain_error`	Signale une erreur de domaine au sens mathématique du terme.
`invalid_argument`	Signale un paramètre invalide.
`length_error`	Signale une tentative de dimensionnement d'un objet tel que celui-ci excéderait sa capacité.
`out_of_range`	Signale une tentative d'accès hors d'un intervalle valide.
`runtime_error`	Devrait servir de parent aux exceptions représentant des erreurs qu'on présume détectables seulement lors de l'exécution d'un programme.
`overflow_error`	Signale un dépassement de la capacité supérieure d'un type au sens arithmétique du terme.
`underflow_error`	Signale un dépassement de la capacité inférieure d'un type au sens arithmétique du terme.
`range_error`	Signale un débordement d'intervalle au ens général dans des calculs.

Ces types sont organisés de manière hiérarchique, par héritage, ce qui signifie qu'un bloc `catch` se saisissant d'une instance de `runtime_error` attrapera aussi une exception de tout type dérivé, et que `catch(exception)` attrapera toute exception standard.



Notez que si un sous-programme peut lever plusieurs types d'exceptions et si le code client désire les traiter de manière spécifique, il est important que les blocs `catch` soient placés du plus spécifique au plus général, du fait que ces blocs seront testés dans l'ordre et que la première clause convenable consommera l'exception levée.

```
try { /* ... */ }
catch (range_error&) { /* ... */ }
catch (exception&) { /* ... */ }
catch (runtime_error&) {
   // jamais atteint!
}
```

De même, certaines exceptions standards sont levées lors des opérations normales de la bibliothèque standard, mais vous pouvez les utiliser dans vos propres programmes. La plupart de ces types dérivent de `std::exception` et sont dans l'espace nommé `std`, incluant `ios_base::failure` (en réalité `std::ios_base::failure`).

Type	Déclaration	Sémantique attendue
`bad_alloc`	`<new>`	Signale une erreur d'allocation dynamique de mémoire.
`bad_cast`	`<typeinfo>`	Signale une erreur de transtypage avec `dynamic_cast`.
`bad_exception`	`<exception>`	Signale une situation où une exception levée par une opération s'avère complètement inattendue. Les fonctions `unexpected()` et `set_unexpected()` du standard permettent de contrôler ce qui se produira dans ce cas.
`bad_typeid`	`<typeinfo>`	Signale une tentative d'obtenir la description d'un type (avec l'opérateur `typeid()` en passant par un pointeur nul.
`ios_base::failure`	`<ios>`	Signale une erreur d'accès sur un flux, pour les programmes qui préfèrent gérer des exceptions lors d'entrées/ sorties. Dérive de `runtime_error`.

Manières d'attraper et stratégies connexes

La plupart des règles applicables aux signatures des paramètres passés à un sous-programme sont applicables aux signatures des types dans les blocs `catch`. L'exemple à droite montre qu'il est possible d'attraper une exception par valeur (voir `p0()`) ou par référence (voir `p1()`).

Lorsqu'un bloc `catch` attrape une exception par valeur, l'exception attrapée est une copie de l'exception lancée, ce qui sollicite le constructeur de copie et le destructeur de la variable temporaire générée. Ainsi, attraper une exception par référence est habituellement plus efficace, évitant de créer une temporaire.

Il peut y avoir des cas pathologiques où il est important d'attraper une copie de l'objet lancé. Dans bien des cas, toutefois, attraper une référence est tout à fait convenable.

Notez qu'il est aussi possible de lever une référence, dans la mesure où l'objet référé a une portée excédant celle du sous-programme dans lequel se produit l'exception. La plupart des programmeurs C++ diront, pour fins d'efficacité, « *Throw by value, catch by reference* ».

En C++, il est légal mais fortement déconseillé de lever des exceptions créées dynamiquement (avec `new`), du fait que le code client, où se situe le bloc `catch`, devra alors libérer lui-même la mémoire associée à cette exception. Bien qu'il soit d'usage, dans les langages reposant sur une collecte automatique d'ordures, de créer dynamiquement les exceptions, cette approche n'est pas idiomatique en C++.

```
#include <exception>
#include <iostream>
// ...using...
int f();
void p0() {
   try {
      f();
   } catch (exception &e) {
      cerr << e.what()
           << '\n';
   }
}
void p1() {
   try {
      f();
   } catch (exception &e) {
      cerr << e.what()
           << '\n';
   }
}
```

Exceptions et vie des objets

Représenter les exceptions par des classes vides [POOv00] est une stratégie simple et sage, mais il peut arriver que des exceptions porteuses d'états (qu'il s'agisse d'un message ou de quelque chose de plus précis) soit utile.

La classe `HorsBornes` proposée à droite, par exemple, est capable d'exprimer un message d'erreur décrivant les bornes attendues et la valeur hors bornes en langue française sur un flux de sortie standard; il est possible qu'un programme apprécie cette fonctionnalité, bien qu'en général les considérations de langue et de format des messages soit du ressort du code client, plus conscient des considérations de localisation d'un programme que ne le sont les divers objets qui y habitent.

```
#include <ostream>
class HorsBornes {
   int minv, maxv, val;
public:
   HorsBornes(int minv, int maxv, int val)
      : minv{minv}, maxv{maxv}, val{val}
   {
   }
   friend std::ostream&
      operator<<(std::ostream &os, const HorsBornes &hb) {
         return os << "Valeur " << hb.val << " hors de ["
                   << hb.minv << ".." << hb.maxv << ']';
   }
};
```

Prudence toutefois pour ce qui est du comportement (surtout lors de copies) des objets utilisés pour représenter les exceptions. En particulier, les méthodes d'un objet représentant une exception ne devraient pas lever d'exceptions, car cela mènerait parfois à des situations de récursivité explosives[94].

Réflexion 01.2 : le message contenu dans une exception standard C++ est un `const char *`, pas une `std::string`. Est-ce un bon choix? Voir ***Réflexion 01.2 : exceptions sans effets secondaires***.

[94] Pensez par exemple à un sous-programme appelé ayant manqué de mémoire, et essayant d'allouer dynamiquement (avec `new`) une instance de `exception`.

*Mécanique de dépilage automatique (*Stack Unwinding*)*

La levée d'une exception dans un sous-programme mène à une opération de **dépilage automatique** (en anglais : *Stack Unwinding*). La mécanique de cette opération va comme suit :

- tous les objets ayant été construits automatiquement dans le sous-programme où l'exception est levée seront automatiquement détruits, par appel de leur destructeur s'il existe, et ce dans l'ordre inverse de celui de leur construction;
- **si l'exception est levée dans un constructeur**, alors tous les attributs déjà construits (et tous les parents de l'objet dont la construction vient d'échouer) seront détruits, en ordre inverse de celui de leur construction. L'objet en cours de construction ne sera pas lui-même détruit, n'ayant pas encore été officiellement construit; son destructeur ne sera donc pas appelé!
- **si l'exception est levée dans un destructeur**, l'objet est officiellement détruit (ses attributs et ses parents sont détruits comme dans le cas où une exception est levée dans un constructeur), mais les opérations incomplètes du destructeur ne seront pas réalisées; il peut donc y avoir perte de ressources.

Sous-programmes prudents

Chose que peu de gens savent: il est possible pour tout sous-programme C++ d'être placé dans un bloc `try`. Cette mécanique a ses limites, mais peut être utile à l'occasion.

Examinons tout d'abord cette mécanique à l'aune d'un constructeur. Imaginons une classe telle `Danger`, à droite, qui est susceptible de lever une exception (ici, une instance de la classe `Flute`, au nom fort significatif), et que certaines autres classes pourraient souhaiter initialiser par préconstruction.

```
class Flute {};
class Danger {
   // ...
public:
   Danger(int val) {
      if (!val) throw Flute{};
      // ...
   }
   // ...
};
```

Le constructeur d'une telle classe, disons la classe `X` à droite, pourrait souhaiter réagir à l'exception levée par la construction d'un `Danger` avant de laisser filtrer cette exception vers le code client (et oui, ici, la neutralité est véritablement indiquée car, blocs `try` et `catch` ou non, l'exception aura été levée et l'instance de `X` n'aura pas été construite).

Remarquez la position de la mention `try` juste avant la section listant les initialisations avant construction, de même que celle du bloc `catch` suivant le code du constructeur. Le constructeur entier fait partie du bloc `try`, et que le traitement d'erreur du bloc `catch` s'applique donc au constructeur entier, *incluant tout ce qui y est préconstruit*.

```
class X {
   Danger danger;
public:
   X(int val)
      try : danger{val} {
         // code du constructeur
      } catch (Flute &f) {
         // ... gérer l'exception
        throw f; // ou throw; tout court
      }
};
```

Si le bloc `catch` d'un constructeur ne lève pas d'exception, un `throw;` implicite y est inséré par le compilateur et l'exception attrapée est automatiquement relancée. Il est essentiel que le sous-programme ayant tenté de construire l'objet soit informé de l'échec de la construction (après tout, un objet n'ayant pas été construit n'existe pas et est *de facto* inutilisable).

Il est évidemment possible de lever explicitement une exception d'un type différent du type d'exception attrapée; on évitera d'agir ainsi, règle générale, puisque cela constituerait un bris de neutralité de la part de l'objet dont la construction aura échoué.

Un exemple de constructeur exploitant cette mécanique dans le but de réaliser un traitement d'exceptions de première ligne est celui de la classe `EntierPositif`, à droite.

Lorsque la valeur reçue en paramètre à la construction d'une instance de cette classe n'est pas conforme aux politiques de validation, une exception est levée. Ici, les conceptrices et les concepteurs de la classe ont choisi d'afficher sur le flux d'erreurs standard un message de diagnostic (pour les humains), puis de laisser filtrer l'exception vers le code client (pour le programme).

Notez que le bloc `catch` n'a pas directement accès à la valeur fautive, ne faisant pas partie du constructeur (placé dans le bloc `try`).

L'attribut `val` de l'instance d'`EntierPositif` n'a pas été construit, dû à l'exception (pas plus que l'instance d'`EntierPositif`, d'ailleurs), ce qui fait que le bloc `catch` n'a pas accès à cet attribut. Ceci explique le recours à une exception spécialisée, `ValeurNegative`, spécialisant (par souci de conformité) `range_error` et contenant un état décrivant la valeur fautive en question. Sans cet état, le bloc `catch` ne pourrait donner que de l'information floue (écrire *la valeur est négative*, sans indiquer quelle est précisément cette valeur).

```cpp
#include <stdexcept>
#include <iostream>
// ...using...
class ValeurNegative : public range_error {
   int val;
public:
   ValeurNegative(int val)
      : range_error{"Valeur négative"},
        val{val} {
   }
   int valeur() const noexcept {
      return val;
   }
};
class EntierPositif {
   int val;
   static int valider(int val) {
      if (val < 0)
         throw ValeurNegative{val};
      return val;
   }
public:
   EntierPositif(int val)
      try : val{valider(val)} {
      } catch (ValeurNegative &v) {
         cerr << v.what()   << " : "
              << v.valeur() << '\n';
         throw;
      }
};
```

La stratégie par laquelle le bloc `try` constitue le code complet d'un sous-programme peut être étendue à tous les sous-programmes.

Ainsi, le code proposé ci-dessous est légal. La fonction `main()` invoque la procédure `f()` en lui passant deux paramètres, puis `f()` invoque `div_ent()` avec ces paramètres mais dans un bloc `try` suivi d'un bloc `catch` en vue d'attraper les cas de division par zéro.

Comme dans le cas des constructeurs implémentés de cette manière, le bloc `catch` n'a accès à aucune des variables internes au bloc `try`. Si le bloc `catch` est atteint, cela signifie que l'exécution de `f()` a échouée suite à la levée d'une exception, et la seule information disponible à ce stade est ce que contient l'exception captée.

Cette stratégie est moins utile qu'il n'y paraît. Entre autres, elle ne permet pas de nettoyer correctement les ressources prises en charge manuellement dans le bloc `try` puisque les variables de ce bloc sont inaccessibles dans le bloc `catch`[95].

Certains iront jusqu'à placer `main()` tout entier dans un bloc `try` suivi d'un `catch(...)` pour réaliser un traitement de dernière chance avant une fermeture imprévue de programme. Il faut que le code dans ce bloc `catch` soit très simple : la raison de l'exception pourrait être `std::bad_alloc`!

```
#include <iostream>
using namespace std;
class DivParZero {};
int div_ent(int num, int denom) {
   if (denom == 0) throw DivParZero{};
   return num/ denom;
}
void f(int num, int denom)
   try {
      cout << div_ent(num, denom) << endl;
   } catch (DivParZero &) {
      cerr << Zut\n";
   }
int main() {
   for (int num, denom; cin >> num >> denom; )
      f(num, denom);
}
```

[95] Raison de plus pour appliquer l'idiome `RAII`, couvert plus haut.

Les cas qui nous échappent

Il nous reste à discuter de ce qui se produit lors de la fin d'un programme C++, que cette fin soit due à une exception échappée ou qu'il s'agisse de la fin normale du programme.

La fonction `std::uncaught_exception()`

Il existe un prédicat standard nommé `std::uncaught_exception()` qui retourne `true` seulement si une exception a été levée mais n'a pas encore été traitée. Cette fonction est déclarée dans `<exception>`. Typiquement, un destructeur pourra savoir qu'il est invoqué suite à un dépilage automatique (voir plus haut) s'il invoque cette fonction et qu'elle retourne `true`.

Herb Sutter, dans [MExcCpp], fait clairement la démonstration que cette fonction bien qu'informative, ne peut être utilisée de manière sécuritaire pour prendre des décisions quant aux moments appropriés pour lever ou non des exceptions. À moins d'avoir une intuition très novatrice, il est hautement probable qu'utiliser cette fonction, bien que ce soit possible, soit une mauvaise idée.

La fonction `std::unexpected()`

Une fonction spéciale, `std::unexpected()`, déclarée dans `<exception>`, sera invoquée par la mécanique du langage si une fonction liste explicitement les cas d'exceptions possibles dû à son exécution, dans le cas où son exécution mènerait malgré tout à la levée d'une exception qui n'apparaît pas dans cette liste.

> Depuis C++ 11, les spécifications d'exceptions sont obsolètes [POOv00]. Ce qui suit est à titre indicatif; toutefois, `std::unexpected()` peut encore se produire à tout moment où deux exceptions sont actives.

L'exemple à droite illustre cette mécanique. Souvenons-nous qu'une fonction ne listant pas de cas d'exception possibles, comme `g()`, peut lever n'importe quel type d'exception. Ici, `f()` pense ne pouvoir lever que `Ouille`, or dû à `g()` elle pourrait aussi lever `Ayoye`.

Si `g()` est invoquée par `f()` avec un `n` de valeur paire, alors `f()` contreviendra à son contrat et la fonction globale `std::unexpected()` sera invoquée.

```
class Ouille{};
class Ayoye {};
int g(int);
void f(int n) {
   if (g(n) != 0)
      throw Ouille{};
   // ...
}
int g(int n) {
   if (n % 2 == 0)
      throw Ayoye{};
   return n + 1;
}
```

> Il est possible de gérer soi-même ce qui se produit à l'appel de `std::unexpected()` en installant un gestionnaire maison à l'aide de `std::set_unexpected()`.
>
> Ceci ne permet en pratique que des opérations de dernier recours : le programme est, à ce stade, dans un état instable.

La fonction `std::exit()`

⇒ Un programme C++ se **complète normalement** par un appel à `std::`**`exit()`**.

La fonction `std::exit()`, déclarée dans `<cstdlib>`, prend en paramètre un entier, qui sera la valeur officielle et effective retournée par le programme C++ qui se complète. Invoquer la fonction `std::exit()` équivaut à utiliser l'opération `return` dans `main()`.

Invoquer la fonction `std::exit()` assure entre autres la bonne destruction des objets globaux, appelant les destructeurs concernés dans les règles de l'art.

La fonction `std::abort()`

⇒ Un programme C++ se **complète anormalement** par un appel à `std::`**`abort()`**.

La fonction `abort()`, déclarée dans `<cstdlib>`, termine brutalement un programme. Le nettoyage normal des ressources du programme, incluant l'appel des destructeurs des objets globaux, n'est pas fait. Un code de retour standard[96] est alors retourné par le programme pour signaler au système d'exploitation la complétion anormale du processus.

On peut invoquer explicitement `std::abort()`, mais un programme bien écrit évitera d'agir ainsi sauf en de graves circonstances. Privilégiez `std::exit()` ou, encore mieux, le retour habituel d'un code d'erreur fait par `main()`.

La fonction `std::terminate()`

⇒ Un programme C++ se **complète anormalement lors d'un problème sérieux de traitement d'exception** par un appel à `std::`**`terminate()`**.

On ne devrait pas appeler `std::terminate()` explicitement. Cette fonction sera invoquée dans le cas où :

- une exception est lancée dans un programme C++ sans être attrapée par ce programme;
- un destructeur génère une exception pendant un dépilage automatique (voir plus haut); ou
- la pile d'exécution d'un programme est corrompue.

Dans chaque cas, on parle donc d'ennuis sérieux, au point de mettre le programme dans un état irrécupérable. La fonction `terminate()` livrée à même le standard se termine par un appel à `abort()`; on peut remplacer la fonction `terminate()` standard en appelant `set_terminate()`; référez-vous à de la documentation technique pour plus de détails.

[96] La valeur `3`, en fait, pour celles et ceux que ça intéresse.

Dans d'autres langages

En Java, les exceptions sont nécessairement des objets. En fait, Java propose une hiérarchie de base pour les cas d'erreurs :

- toute notification de problème susceptible d'être levée doit être un dérivé direct ou indirect de la classe `Throwable`;
- en Java, un `Throwable` peut être levé et capté;
- deux grandes familles de dérivés de `Throwable` existent, soit les classes dérivant de `Exception` et celles dérivant de `Error`;
- les dérivés d'`Error` sont des erreurs sérieuses qu'on ne recommande pas aux programmes de capter. Les dérivés d'`Exception` sont des erreurs dont un programme pourrait raisonnablement espérer récupérer;
- le choix de traiter ou non une erreur (dérivé d'`Error`) est du ressort du programmeur. Le traitement des exceptions (dérivé d'`Exception`) est obligatoire;
- si une méthode est invoquée et si cette méthode est susceptible de lever plusieurs types d'exceptions, alors le bloc `try` peut être suivi de plusieurs blocs `catch` en séquence;
- le premier bloc `catch` pour lequel le type d'exception à attraper correspond au type d'exception levée sera emprunté. Cela signifie qu'il est important d'attraper les exceptions les plus spécifiques avant d'attraper les exceptions les plus générales (comme `Exception` par exemple, qui est un type fourre-tout dans ce cas, un peu comme `catch(...)` en C++).

En C#, une exception dérive de la classe standard `exception` et les blocs `catch`, susceptibles d'être placés en séquence, sont disposés du plus spécifique au plus général, exactement comme en Java. Le message d'une `exception` est placé dans sa propriété `Message` [POOv00].

Un peu comme en Java, les exceptions sont groupées sous deux classes de base que sont `ApplicationException` et `SystemException`. En C#, par contre, les exceptions prédéfinies dérivent de `SystemException` alors que les autres, probablement définies par les programmes, dérivent de `ApplicationException`. La distinction est faite par provenance de la classe plutôt que par la gravité du problème.

Créer votre propre classe d'exception implique faire de cette classe une classe dérivée de `ApplicationException` ou d'un de ses dérivés apparaissant plus approprié au problème relevé et s'en servir à travers les mécanismes du langage.

En `VB.NET`, une exception dérive de la classe standard `Exception` et les blocs `Catch`, susceptibles d'être placés en séquence, sont disposés du plus spécifique au plus général, exactement comme en Java. Les remarques utilisées dans la section portant sur C# ci-dessus s'appliquent aussi dans le cas de `VB.NET`.

Exercices – Série 06

EX00 – Reprenez la classe HorsBornes examinée dans la section ***Exceptions et vie des objets***. Adaptez cette classe pour qu'elle dérive d'une classe d'exception standard de C++ (voir ***Types standards d'exceptions***) et pour faire en sorte qu'elle spécialise correctement la méthode what()? Quelle sera sa classe parent? Comment adapterez-vous HorsBornes pour qu'elle conserve la fonctionnalité existante sans pour autant implémenter de fonctionnalités redondantes?

EX01 – Étant donné le code proposé à droite :

- déterminez s'il est sécuritaire ou non (indice : il ne l'est pas, mais il vous faut comprendre pour quelle raison);
- plus sérieusement : assurez-vous que la BD soit fermée en tout temps par la fonction f(), en utilisant le traitement d'exceptions;
- assurez-vous que la BD soit fermée en tout temps par la fonction f(), en utilisant une approche RAII;
- comparez les deux approches.

```
int ouvrirBD();
void fermerBD(int);
void utiliserBD(int);
void f() {
   auto id = ouvrirBD();
   utiliserBD(id);
   fermerBD(id);
}
```

Appendice 00 – Association, agrégation et composition

Si l'héritage est une relation sur le mode du verbe être (en anglais : *Is-A Relationship*), il existe aussi trois familles de relations OO sur le mode du verbe avoir (en anglais : *Has-A Relationship*). Ces relations ont toutes trois été mentionnées (et utilisées) antérieurement dans les notes de cours, mais le temps est venu de les aborder plus formellement.

Composition

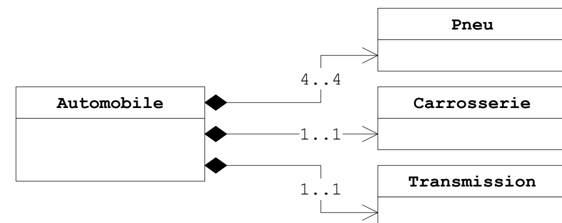
Bref retour sur la composition, que nous utilisons de manière routinière depuis presque le tout début (en effet, qui dit attributs dit habituellement composition, surtout en C++ où les classes peuvent être des types valeurs tout autant que les types primitifs).

Concevoir un objet par composition implique de regrouper sous son égide plusieurs composants de manière à ce que la durée de vie de chaque composant soit délimitée par la durée de vie de l'objet composé. Prenons par exemple une automobile (très simplifiée). Le squelette proposé à droite montre que chaque instance d'`Automobile` sera *composée* d'un certain nombre d'attributs qui sont eux-mêmes des objets.

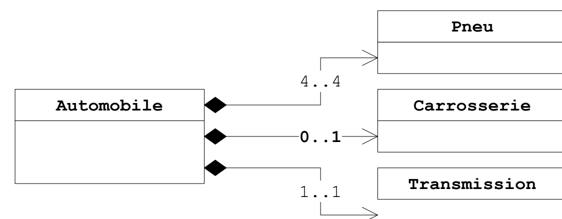
```
// inclusions...
class Automobile {
   enum { NPNEUS = 4 };
   Pneu pneus_[NPNEUS];
   Carosserie coque_;
   Transmission transmission_;
   // etc.
}
```

Chacun de ces attributs commencera à exister *juste avant* la construction de l'instance d'`Automobile` dont il fait partie, et cessera d'exister *juste après* la destruction de celle-ci.

On se souviendra que la notation UML appliquée pour la composition est celle proposée à droite. La cardinalité est exprimée sur chaque lien par un intervalle. Dans ce cas, plutôt simple, la cardinalité est chaque fois fixée (quatre pneus, une carrosserie, une transmission, *etc*.).



Si la carrosserie est allouée dynamiquement mais toujours à l'interne dans `Automobile`[97], et s'il est possible que l'objet se trouve à l'occasion sans carrosserie, alors le schéma UML devient celui à droite. Remarquez la cardinalité de la relation de composition entre `Automobile` et `Carrosserie`.



C'est là la signature d'un design par composition : la durée de vie des *composants* ne peut excéder celle du *composé*. Ceci permet une gestion des durées de vie à même les constructeurs et le destructeur du composé, ce qui est une manière naturelle de procéder selon l'approche objet.

[97] Si chaque instance d'`Automobile` est responsable de faire `new Carrosserie` et de faire le `delete` correspondant avant d'être elle-même détruite.

Agrégation

L'agrégation est une proche cousine de la composition, les deux techniques servant à résoudre des problèmes similaires.

L'agrégation se reconnaît au fait que l'existence des objets constituant l'agrégat[98] peut précéder ou excéder celle de l'agrégat lui-même. Par conséquent, en C++, on implante un agrégat à l'aide de pointeurs ou de références, pour que la destruction de l'objet n'entraîne pas nécessairement celle de ses composants.

À titre d'exemple[99], présumons qu'on ait la classe `Eleve`, offrant entre autres un opérateur d'affectation, un constructeur par défaut et un constructeur par copie, et qu'on veuille rédiger la classe `Groupe`, représentant un groupe d'instances d'`Eleve` (par exemple, les élèves d'un cours donné ou d'une cohorte donnée).

L'implantation ***par composition*** (à droite) pourra se faire à l'aide d'un tableau d'`Eleve`. Ainsi, chaque `Eleve` du tableau aura une vie liée à celle du `Groupe`.

```
#include "Eleve.h"
#include "Incopiable.h"
class Groupe : Incopiable {
   static const int NMEMBRES_DEFAUT;
   Eleve *membres;
   const int nmembres = NMEMBRES_DEFAUT;
   // ...
public:
   Groupe() {
      membres=new Eleve[nmembres];
   }
   Groupe(int nb) : nmembres{nb} {
      membres=new Eleve[nmembres];
   }
   void SetEleve(const Eleve &e, int n) {
      membres[n]= e;
   }
   Eleve eleve(int n) const {
      return membres[n];
   }
   // ... reste de la Sainte-Trinité, etc.
   ~Groupe() {
      delete[] membres_;
   }
};
```

La méthode `SetEleve()` insérera dans le tableau une copie de l'instance d'`Eleve` passée en paramètre, alors que la méthode `eleve()` accèdera en lecture à un `Eleve` en particulier.

L'approche par composition convient dans la mesure où les instances d'`Eleve` sont utiles parce qu'elles font partie du `Groupe`. On remarquera par contre que si chaque `Eleve` fait partie de plusieurs instances différentes de `Groupe`[100], on se retrouvera avec certains problèmes.

L'un de ces problèmes est que, chaque `Groupe` utilisant une copie distincte de l'instance d'`Eleve`, modifier l'une des copies ne modifie pas automatiquement les autres. L'implantation par agrégation, elle, pourra se faire par un tableau de `Eleve*` (pointeurs d'`Eleve`), en présumant que chaque `Eleve` ait été créé avant de se joindre au `Groupe`.

---

[98] À ne pas confondre avec le terme *Aggregate* du langage C, qui indique toute chose faite de parties (en ce sens, tous les `struct` et toutes les classes se qualifieraient).

[99] Remarquez que l'implantation qui sera proposée ici est très simple (simpliste, même) et est déficiente sur le plan de la sécurité (validation des tailles et index, entre autres). L'objectif n'est pas de présenter une classe blindée, mais bien de nous aider à distinguer la composition de l'agrégation.

[100] Si un même `Eleve` faisait partie de deux cours distincts, d'une cohorte, d'un club de philatélie, et d'un regroupement musical, par exemple.

La vie d'une instance d'`Eleve`, si on implante `Groupe` comme un agrégat, aura probablement commencé avant que le `Groupe` n'ait été créé.

Notez la syntaxe de la déclaration et de la création de `membres_`. Ici, le type de cet attribut est **`Eleve**`**, donc un *pointeur de pointeur d'`Eleve`*.

Plutôt qu'allouer dynamiquement un tableau d'`Eleve`, on allouera un tableau d'**`Eleve*`**, et on y déposera des pointeurs d'`Eleve` ayant une existence qui leur est propre hors du `Groupe`.

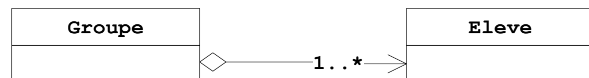
Conséquemment, les méthodes `eleve()` et `SetEleve()` manipulent des pointeurs d'`Eleve`[101], pas des copies, et les constructeurs (seul celui par défaut est présenté ici) initialisent le tableau `membres_` avec des pointeurs nuls (dans l'approche par composition, un constructeur par défaut d'`Eleve` jouait ce rôle).

```
#include "Eleve.h"
#include "Incopiable"
class Groupe : Incopiable {
   static const int NMEMBRES_DEFAUT;
   Eleve **membres;
   const int nmembres = NMEMBRES_DEFAUT;
public:
   Groupe() {
      membres = new Eleve*[nb_membres]{};
   }
   // ...
   void SetEleve(Eleve *e, int n) {
      membres[n] = e;
   }
   Eleve *eleves(int n) noexcept {
      return membres[n];
   }
   const Eleve *eleve(int n) const noexcept {
      return membres[n];
   }
   ~Groupe() {
      delete[] membres_;
   }
};
```

Notez aussi le destructeur, qui détruit le tableau mais pas les instances d'`Eleve` vers lesquelles ce tableau pointe. *La responsabilité de détruire les instances constituant l'agrégat ne revient pas clairement à l'agrégat*[102]. Les diverses instances ayant été créées *a priori*, il peut être malaisé de les détruire ici (qui sait à qui d'autre ces instances peuvent servir? Qui sait même si elles ont été allouées dynamiquement?).

La notation `UML` pour la composition et pour l'agrégation n'est pas la même. On utilise un carreau blanc pour identifier qu'une entité sert d'agrégat pour une autre. Avec la composition, un carreau noir doit être utilisé.

Dans l'exemple à droite, tout `Groupe` sert d'agrégat pour au moins un `Eleve`.



[101] En effet, si le pointeur retourné n'était pas `const`, donc protégé à l'appel contre les accès en écriture, la méthode représenterait un bris de sécurité et ne pourrait plus être `const`!

[102] Il faut, pour bien faire, penser au préalable à une stratégie pour que la vie des éléments d'un agrégat soit correctement gérée, du moins pour un langage comme C++ où il n'existe pas de mécanisme de collecte automatique des ordures. Dans C++ `11`, la classe `std::shared_ptr` simplifiera cette problématique.

Agrégation composition et délégation

On appliquera parfois la technique de l'enrobage, vue précédemment, à des agrégats de la même manière qu'on pouvait l'appliquer à des composés. Les règles sont essentiellement les mêmes dans les deux cas. Notez que le terme savant de cette tactique est la **délégation**. Ne confondez pas délégation avec les ***délégués*** (`delegates`), un concept semi `OO` sur lequel nous reviendrons plus loin. L'agrégat délègue le traitement d'un appel de méthode à l'un de ses objets constitutifs.

Un exemple d'application de la paire de techniques agrégation/ délégation serait celui proposé à droite.

La classe `CreateurNom` reçoit à la construction deux références constantes, soit une sur un générateur de prénom et une sur un générateur de noms de famille. Elle ne connaît pas la stratégie (ou la langue!) utilisée pour générer des noms, et il en va de même pour les prénoms.

En retour, elle est capable de générer une paire $\{Nom, Prénom\}$ sur demande en combinant les stratégies de ces deux générateurs.

Le fait qu'elle délègue une partie de ses opérations à certains des membres de l'agrégat qu'elle constitue représente une application concrète de la délégation.

L'agrégation permet des schémas de conception reconnus, comme celui nommé *Whole-Part*.

```
#include <string>
struct GenNom {
   virtual std::string generer() const = 0;
   virtual ~GenNom() = default;
};
struct GenPrenom {
   virtual std::string generer() const = 0;
   virtual ~GenPrenom() = default;
};
class CreateurNom {
   const GenNom &gnom;
   const GenPrenom &gprenom;
public:
   CreateurNom(const GenNom &gn,
               const GenPrenom &gp)
      : gnom{gn}, gprenom{gp}
   {
   }
   std::string generer() const {
      return gnom.generer() + ", " +
             gprenom.generer();
   }
}
```

Quand un agrégat partage des membres avec d'autres objets, le problème de savoir qui sera responsable de la destruction de l'objet partagé survient. Nous y reviendrons dans notre discussion des ***Pointeurs intelligents et objets partageables*** [POOv02].

Association

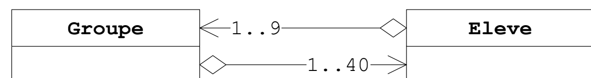
Une troisième relation, plus simple mais tout aussi fondamentale, est l'**association**, par laquelle on entend tout lien entre deux objets qui permet à l'un d'envoyer des messages à l'autre (concrètement : d'appeler des méthodes de l'autre).

Conceptuellement, toute relation générique entre deux classes ou deux instances est une association. La composition, l'agrégation et l'héritage sont des raffinements du concept général d'association. L'association est si générale, en fait, qu'elle n'apparaît pas comme concept dans la plupart des langages.

Les relations de composition, d'agrégation et d'association son directionnelles. Plus haut, `Automobile` est composée (en partie) d'instances de `Pneu`. Cela signifie un savoir qu'a `Automobile` quant à `Pneu`. L'inverse n'est pas nécessairement vrai : rien n'indique que `Pneu` doive connaître `Automobile`.

Il est aussi possible que des relations existent dans les deux directions. Si un `Groupe` connaît ses instances d'`Eleve`, il se peut aussi qu'un `Eleve` connaisse ses instances de `Groupe`. Les deux relations sont probablement des relations d'agrégation, et sont distinctes l'une de l'autre.

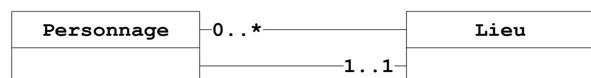
Elles n'ont pas nécessairement la même cardinalité : un `Groupe` peut, peut-être, contenir jusqu'à `40` instances d'`Eleve`, alors qu'un `Eleve` peut, peut-être, connaître jusqu'à `9` instances de `Groupe`.



L'association dénote toute relation entre deux objets qui permette à ces objets de communiquer, dans un sens ou dans l'autre (ce que nous indiquent les cardinalités), qui ne soit ni héritage, ni composition, ni agrégation.

Prenons par exemple, un `Personnage` et un `Lieu`. Les deux entités sont logiquement distinctes. Un `Personnage` ne possède pas le `Lieu` mais peut être de passage, et que le `Lieu` n'est pas non plus responsable du `Personnage`. On pourrait prétendre qu'un `Personnage` visite un `Lieu` à la fois (pas plus, pas moins), et qu'un `Lieu` est en tout temps visité par un nombre arbitraire d'instances de `Personnage` (`0, 1, 2,` …).

La relation d'association se présente, en notation `UML`, comme dans le schéma à droite. Ni carreaux, ni flèches, et une cardinalité du côté de l'objet de la relation (pas du côté du sujet).



Appendice 01 – De l'importance des constantes dans le modèle OO

L'importance des constantes est une chose qu'on enseigne habituellement aux étudiantes et aux étudiants en informatique lors des tout premiers cours de programmation et à laquelle trop de gens (certains enseignants inclus) n'offrent pas suffisamment de considération.

Merci à celles et ceux qui ont relu ce texte et ont suggéré des corrections ou des raffinements (incluant un merci particulier à ***Pierre Prud'homme*** qui fut particulièrement prolifique et pertinent).

L'argument traditionnel pour utiliser des constantes se détaille à peu près comme suit :

- **les constantes clarifient la sémantique du code**. Ceci est utilisé pour indiquer aux étudiantes et aux étudiants en début de formation que le littéral `9` peut avoir plusieurs sens dans un programme (nombre d'individus dans un groupe, carré de trois, prix d'une pizza, ...) alors que `NB_PLANETES` a un sens beaucoup plus clair et univoque;
- **les constantes facilitent l'entretien du code**. Dans le cas d'un programme traitant du système solaire par exemple, avec le retrait de Pluton de la liste des planètes du système solaire en `2006`, il aurait fallu traverser le code d'un programme utilisant le littéral `9` pour examiner chaque occurrence de ce littéral. Il aurait alors fallu évaluer, pour chaque occurrence, si ce `9` symbolise le nombre de planètes en orbite autour de notre soleil alors que, si la constante symbolique `NB_PLANETES` avait été utilisée, il aurait suffi de changer sa valeur et de retoucher les structures de données qui en dépendent. Cette pratique est lourde et sujette à erreur;
- **les constantes constituent une forme de documentation par déclaration d'intention**. Savoir qu'une donnée est constante constitue une forme d'information pertinente pour les programmeurs et aide à leur compréhension des algorithmes. Après tout, quoi de plus suspect que de rencontrer une donnée susceptible d'être constante et de constater qu'il s'agit d'une variable. La question se pose alors : pourquoi?
- **les constantes donnent des garanties d'invariance**. Savoir que `PI` est une constante plutôt qu'une variable permet de se fier à sa valeur, qui ne sera pas modifiée de manière hostile ou par inadvertance dans le programme; par conséquent
- **les constantes permettent à un compilateur de générer du meilleur code**. Bien que les compilateurs ne procèdent pas tous aux mêmes optimisations et bien que certains négligent certaines opportunités d'optimisation en apparence évidentes, il reste que plus un compilateur possède d'information (pertinente) sur un programme et meilleures sont ses chances de générer du code de qualité. Savoir qu'une donnée est constante permet au compilateur de procéder à certaines optimisations plus agressives que sur des variables puisque le fruit de certains tests faites sur cette donnée (par exemple : est-elle négative?) sont susceptibles d'être soit toujours vrais, soit toujours faux. Sachant cela, même s'il est possible que votre compilateur ne profite pas toujours à plein de l'information quant à la constance d'une donnée, il n'y a pas de raison de ne pas lui donner l'opportunité de le faire.

En général, *si une donnée peut être marquée constante, alors elle devrait l'être*. On peut ne rien y gagner, mais il est certain qu'on n'y perdra rien. C'est là une maxime utile à retenir, mais une maxime sans compréhension est d'une utilité limitée. Il y a beaucoup d'autres raisons pour avoir recours aux constantes, en particulier dans un monde `OO`, et c'est sur ces cas fort intéressants que nous poserons notre regard ici.

Définition de l'encapsulation

Ma définition de l'encapsulation dépasse le seuil des mots privé, protégé et public qu'on retrouve dans la plupart des langages de programmation `OO` et se situe à un niveau plus philosophique. Elle se décline en deux temps :

- un objet doit être responsable de son intégrité du début à la fin de sa vie; et
- un objet doit être capable de faire en sorte de nettoyer les traces de son passage dans un système. Ses effets secondaires[103] doivent être délibérés et il ne doit pas entraîner de fuites de ressources.

Contrairement à ce que les gens pensent souvent, il est très difficile d'assurer le respect du principe d'encapsulation exprimé en ces termes en Java et dans les langages `.NET`.

La clause sur les effets de bord y est difficile à mettre en place malgré la présence d'un moteur de collecte automatique d'ordures, ou même à cause de lui : Java n'offre aucune garantie d'appel à la méthode de nettoyage `finalize()` (elle n'est souvent pas appelée du tout) et que les langages `.NET` invoquent `Dispose()` ou son équivalent mais sans qu'aucune référence prise en charge dans l'objet en cours de nettoyage ne puisse être considérée valide, limitant drastiquement la possibilité de bien nettoyer le code. Intégrer ces langages à toute ressource externe exige la mise en place de schémas de conception ou d'idiomes de programmation (p. ex. : `IDisposable` sous `.NET`) et l'insertion de code pour libérer les ressources manuellement, comme par exemple appeler `close()` sur les fichiers ou insérer des blocs `finally` dans le traitement d'exceptions.

Ces reliquats d'une époque pré-`OO` sont la conséquence directe de la collecte d'ordures prise en charge, qui doit fonctionner même quand deux objets se réfèrent mutuellement l'un et l'autre : il s'agit de l'inverse d'une médaille *a priori* fort jolie, qui nous rappelle qu'il n'y a rien de gratuit et que ce qui semble simplifier le code dans certains cas devient, à l'inverse, un cauchemar d'entretien dans le monde intégré et interopérable vers lequel nous tendons au moment d'écrire ceci. Prudence : la collecte automatique d'ordures n'est pas en soi une mauvaise chose. Elle règle des problèmes et permet certains styles de programmation, mais ce n'est pas une panacée.

La principale cause de l'incapacité d'implémenter l'encapsulation *au sens des garanties d'intégrité* en Java et dans les langages `.NET` tient de l'incapacité d'exprimer des constantes dans ces langages. C'est ce que nous démontrerons ici, en apportant les nuances qui s'imposent pour les classes immuables et en expliquant le prix à payer pour faire du code robuste sans avoir accès à la possibilité d'exprimer quelque chose d'aussi fondamental que des constantes.

[103] En premier lieu, j'avais utilisé *effets de bord*, mais *effets secondaires* est plus adéquat. Un éminent collègue m'a suggéré effets secondaires et effets indésirables en privilégiant effets indésirables; cependant, la plupart des effets secondaires découlant de la vie d'un objet me semblent désirés (après tout, dans la plupart des cas, un objet qui consomme des ressources le fait volontairement) alors j'ai privilégié effets secondaires, plus près de ma pensée ici.

Parenthèse : mais je n'ai jamais eu de problèmes, moi!

Je rencontre souvent des gens qui me disent ne jamais vivre de problèmes d'intégrité de code en Java et en C#, plusieurs étant des programmeuses et des programmeurs de qualité. Je suis certain que cela correspond à leur expérience de développement. Cependant, l’absence dans leur vécu de difficultés comme celles évoquées ici tient surtout au contexte dans lequel leur code a été développé et utilisé.

Certains problèmes, même parmi les plus fondamentaux, n’apparaissent au grand jour que lorsqu’un programme est manipulé par d’autres que ceux qui l’ont écrit. C'est l’une des raisons pour lesquelles j'aime parler, dans de telles circonstances, de *code de bibliothèque*. J’entends par cette expression du code conçu par certains et utilisé par (beaucoup!) d’autres, donc qui doit être aussi robuste que possible tout en permettant d’atteindre des niveaux élevés de performance. En effet, si un module est sollicité massivement dans un programme, comme du code de bibliothèque risque de l’être, alors il doit être extrêmement robuste et aussi rapide que possible.

À mon avis, tout objet doit être conçu comme s’il pouvait évoluer de manière à devoir se conformer aux contraintes propres à du code de bibliothèque. Une fois un objet publié, il est susceptible d’être utilisé un peu partout et de servir comme point névralgique dans un système complexe. C’est pourquoi il faut au moins lui offrir la possibilité d'évoluer vers le niveau de qualité attendu pour du code de ce niveau.

Le respect de l’encapsulation est crucial en ce sens. Toute brèche dans la barrière d’abstraction d’un objet (par exemple l'introduction d'un attribut protégé ou public) brise immédiatement la capacité qu’a une instance d’une classe `X` d’être garante de sa propre intégrité, du fait qu'un tiers (instance d’une classe dérivée ou autre objet du système, selon le cas) devient capable de modifier l'état de `X` à son insu.

Beaucoup de (bon!) code Java et `.NET` se trouve en circulation aujourd’hui, mais une trop grande partie de ce code est du gruyère du point de vue de l’encapsulation. La principale cause n'est pas la qualité des programmeuses et des programmeurs mais bien la difficulté, beaucoup plus grande que ne le pensent la plupart des gens, d’écrire du code à la fois robuste et rapide dans ces langages.

Cas d’espèce

Imaginons une classe `Rectangle` offrant un ensemble de services, en particulier celui permettant à un `Rectangle` de se dessiner. Certains choix d'implémentation doivent être faits mais ne devraient pas affecter l'interface par laquelle un `Rectangle` est utilisé.

Par exemple, si un `Rectangle` offre des services permettant d'en connaître la hauteur et la largeur, il importe peu pour un client du `Rectangle` de savoir si ces données sont calculées à partir des coordonnées de deux coins opposés du `Rectangle` ou si elles sont entreposées sous forme d'attributs.

Nous débuterons notre discussion en présumant les services suivants pour la classe `Rectangle`. Notez que même l'acte d'établir une liste de services est un geste politique dont la portée est sujette à discussion (certains choix de services proposés ci-dessous sont porteurs de conséquences distinctes selon les langages).

Nom	Vocation	Peut modifier l'objet	Remarques
Constructeur par défaut	Initialiser un `Rectangle` typique.	Oui	Le sens de `Rectangle` typique est politique et dépend des besoins et des règles locales. Cette méthode ne dépend d'aucun apport externe et devrait donc tendre à réduire les validations au minimum.
Constructeur paramétrique	Initialiser un `Rectangle` à partir de paramètres suppléés par le sous-programme procédant à son instanciation.		Cette méthode est susceptible de se faire offrir des paramètres incorrects et doit donc procéder à une validation exhaustive de ses intrants.
Constructeur par copie	Initialiser un `Rectangle` à partir d'un autre `Rectangle` de manière à ce que, suite à la construction, la copie et l'original soient identiques au sens d'une comparaison de contenu.		Cette méthode est un support mécanique important en C++ du fait que ce langage exploite beaucoup les objets selon une sémantique par valeur. Ici, `Rectangle` étant un type valeur, son implémentation sera implicite.
Méthode de clonage	Demander à un `Rectangle` d'instancier dynamiquement une copie de lui-même et de la retourner.	Non	En C++, on n'en voudra pas, `Rectangle` n'étant pas polymorphique, mais dans des langages comme Java, cette méthode est très importante.
Destructeur	Libérer les ressources qu'une instance de `Rectangle` possède avant que celle-ci ne soit détruite.	Oui	`Rectangle` étant un type valeur, son implémentation sera implicite. Les divers langages `OO` commerciaux ont des approches très différentes sur ce sujet.
Accesseurs pour la hauteur et la largeur	Permettre d'interroger un `Rectangle` pour en connaître la hauteur et la largeur.	Non	Peuvent s'implémenter sous forme de propriété dans certains langages, en particularité les langages `.NET`, mais cela n'ajoute rien et n'enlève rien au propos alors nous passerons outre.
Mutateur pour la hauteur et la largeur	Permettre de tenter de modifier la hauteur ou la largeur d'un `Rectangle`.	Oui	Peuvent s'implémenter sous forme de propriété dans certains langages, en particularité les langages `.NET`, mais cela n'ajoute rien et n'enlève rien au propos alors nous ne nous en occuperons pas ici. Dans les langages où les objets sont basés sur des indirections et qui ne permettent pas les objets constants, il est souvent sage d'omettre les mutateurs. Il est utile, dans une optique de performance, de décliner les mutateurs en deux catégories : les mutateurs privés, optimisés pour la vitesse et qui ne procèdent à aucune validation, et les mutateurs protégés et publics, blindés contre les assauts de tiers hostiles.
Méthode `dessiner()`	Demander à un `Rectangle` de se projeter sur un flux de sortie.	Non	Nous présumerons que le flux de sortie en question sera la console pour que le code reste simple.
Autres méthodes[104]	Selon les besoins : comparer deux instances de `Rectangle` entre elles; les classer en ordre croissant selon un critère d'ordonnancement donné; déplacer une instance de `Rectangle` sur un plan; etc.	Variable	Laissé à votre imagination. La gamme des possibles est un ensemble ouvert.

Portez attention à la colonne **Peut modifier l'objet**, indiquant s'il est raisonnable d'accepter que l'invocation de cette méthode provoque une modification à l'état de l'instance propriétaire.

[104] Opérateurs relationnels ou équivalent; opérateur d'affectation ou équivalent; méthodes de translation ou de rotation; *etc.*

Vision C++ : constantes sur une base instance

Un exemple de code C++ couvrant les diverses opérations proposées pour la classe `Rectangle` ci-dessus serait le suivant. Notez que l'implémentation proposée implémente des mutateurs, pour les fins de la discussion, mais en pratique on aurait sans doute préféré ne pas offrir ces méthodes et privilégier l'affectation comme mode de modification d'un objet. Aussi, en C++, `Rectangle` sera un type valeur et n'offrira pas de méthode de clonage, mais si une telle méthode était offerte, elle serait qualifiée `const`.

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
//
// Rectangle.h
//
#include <iosfwd>
class Rectangle {
public:
   // type interne public
   using size_type = short;
private:
   // une implémentation possible (parmi plusieurs)
   size_type hauteur_ = HAUTEUR_DEFAUT,
             largeur_ = LARGEUR_DEFAUT;
   enum {
      LARGEUR_MIN =  1, HAUTEUR_MIN =  1,
      LARGEUR_MAX = 80, HAUTEUR_MAX = 24,
      LARGEUR_DEFAUT = 5, HAUTEUR_DEFAUT = 3,
   };
   static bool est_hauteur_valide(size_type hauteur) noexcept {
      return HAUTEUR_MIN <= hauteur && hauteur <= HAUTEUR_MAX;
   }
   static bool est_largeur_valide(size_type largeur) noexcept {
      return LARGEUR_MIN <= largeur && largeur <= LARGEUR_MAX;
   }
public:
   Rectangle() = default;
   Rectangle(size_type hauteur, size_type largeur)
      : largeur_{valider_largeur(largeur)}, hauteur_{valider_hauteur(hauteur)}
   {
   }
   //
   // Rectangle est un type valeur; la Sainte-Trinité (constructeur de copie,
   // affectation, destruction) implicite s'applique
   //
   // accesseurs publics -- interface de consultation
   // accesseurs de premier ordre
   size_type largeur() const noexcept {
      return largeur_;
```

```
   }
   size_type hauteur() const noexcept {
      return hauteur_;
   }
   // accesseurs de second ordre
   size_type perimetre() const noexcept {
      return 2 * (largeur() + hauteur());
   }
   size_type surface() const noexcept {
      return largeur() * hauteur();
   }
   // mutateurs publics -- interface de modification contrôlée
   // versions publiques et blindées. Pourraient être omises
   void SetHauteur(size_type hauteur) {
     SetHauteurBrut(valider_hauteur(hauteur));
   }
   void SetLargeur(size_type largeur) {
      SetLargeurBrut(valider_largeur(largeur));
   }
private:
   // mutateurs privés -- interface de modification primitive
   // versions privées et rapides
   void SetHauteurBrut(size_type hauteur) noexcept {
      hauteur_ = hauteur;
   }
   void SetLargeurBrut(size_type largeur) noexcept {
      largeur_ = largeur;
   }
   static size_type valider_hauteur(size_type hauteur) {
      if (!est_hauteur_valide(hauteur)) throw TailleInvalide{};
      return hauteur;
   }
   static size_type valider_largeur(size_type largeur) {
      if (!est_largeur_valide(largeur)) throw TailleInvalide{};
      return largeur;
   }
public:
   // méthodes de projection sur un flux (sans paramètre : projection à la console)
   void dessiner() const;
   void dessiner(std::ostream&) const;
   // ...plusieurs autres méthodes (laissez-vous aller)...
};

// projection sur un flux arbitraire
std::ostream& operator<<(std::ostream&, const Rectangle&);
#endif
```

Le code est évidemment très semblable à celui utilisé dans [POOv00].

```
//
// Rectangle.cpp
//
#include "Rectangle.h"
#include <iostream>
using namespace std;
void Rectangle::dessiner() const {
   dessiner(cout);
}
void Rectangle::dessiner(ostream &os) const {
   for (size_type i = 0; i < hauteur(); ++i) {
      for (size_type j = 0; j < largeur(); ++j)
         os << '*';
      os << endl;
   }
}
ostream& operator<<(ostream &os, const Rectangle &r) {
   r.dessiner(os);
   return os;
}
```

Réaliser la constance d'un objet

Remarquez la présence dans le code ci-dessus de garanties de constance, indiquées par la présence du mot clé `const`. Ce mot sert à plusieurs sauces mais dans une optique précise. À titre de rappel :

- placé devant le type d'un objet (incluant les données de types primitifs), incluant le cas où l'objet est un paramètre à un sous-programme, ce mot indique que le compilateur doit interdire toute tentative de modification sur cet objet (hormis au moment de sa construction et de sa destruction). Les tentatives de modification proscrites incluent utiliser l'objet à gauche d'une affectation, passer l'objet de manière indirecte (par référence ou par adresse) à tout sous-programme ne garantissant pas qu'il restera intact ou encore invoquer toute méthode de cet objet qui ne se porte pas elle-même garante de l'intégrité de l'objet. C'est cette transitivité de la clause de constance qui permet au compilateur de l'implémenter efficacement;
- placé suite à la parenthèse fermante de la liste des paramètres d'une méthode d'instance, ce mot indique que la méthode garantit ne pas changer l'état de l'instance à laquelle elle appartient.

> Notez qu'une méthode d'instance reçoit un paramètre implicite (`this`) et que le mot clé `const` suivant la parenthèse fermante d'une liste de paramètres passés à une méthode d'instance est simplement appliqué à `this`. Il y a, au fond, une seule règle, pas deux.

Sachant quelles sont ces quelques règles, il devient clair qu'un objet constant est un objet sur lequel on ne peut réaliser que des opérations garantissant la non modification de ses états. Plus concrètement, dans le code suivant :

- deux instances de `Rectangle`, `r0` et `r1`, sont initialisées avec des contenus identiques mais sont des objets distincts, à ceci près que `r0` est `const` et que `r1` ne l'est pas;
- les opérations respectant la constance de `r0` sont légales lorsqu'elles sont appliquées à cet objet. Ainsi, `g(r0)` et `r0.dessiner()` sont toutes deux légales puisqu'elles respectent le fait que `r0` soit `const`, mais `f(r0)` et `r0.SetHauteur()` sont toutes deux illégales car elles sont susceptibles de tenter de modifier `r0`;
- en retour, les opérations sont toutes valides sur `r1`, qu'elles en garantissent la constance ou non, puisque `r1` est un `Rectangle` auquel on n'a pas ajouté la contrainte de constance;

```
#include "Rectangle.h"
void f(Rectangle&);
void g(const Rectangle&);
int main() {
   const Rectangle r0{3, 9};
   Rectangle r1 = r0;
   f(r0); // illégal!
   f(r1); // ok
   g(r0); // ok
   g(r1); // ok
   r0.SetHauteur(5); // illégal!
   r1.SetHauteur(5); // ok
   r0.dessiner();    // ok
   r1.dessiner();    // ok
}
```

- il n'y a aucun coût en espace ou en temps à qualifier un objet de constant. Toutes les validations peuvent être réalisées à la compilation.

> Une opération garantissant la constance d'un objet est applicable aux objets constants comme à ceux qui ne le sont pas. Une opération ne garantissant pas la constance d'un objet n'est applicable qu'aux objets qui ne sont pas constants. Conséquemment, *si une garantie de constance peut être donnée à une opération ou à un paramètre, il est sage de le faire*.

À ce stade du développement, il est possible que vous ne soyez pas convaincu(e) de l'intérêt d'avoir un objet constant. Le `Rectangle` qu'on souhaite dessiner à la console semble être un cas trop simple pour qu'on juge pertinent de vouloir en garantir la constance.

Cependant, imaginons une classe comme la classe `Bouton` ci-dessous, et imaginons que la dimension d'un `Bouton` soit un `Rectangle`. La méthode d'instance `dimension()` de `Bouton` y est légitimement marquée `const` du fait qu'elle retourne une copie du `Rectangle` représentant la dimension d'un bouton, chose inoffensive pour le `Bouton`.

```
#include "Rectangle.h"
class Bouton {
   // ...
   Rectangle dimension_;
   // ...
public:
   // ...
   Rectangle dimension() const noexcept {
      return dimension_;
   }
   // ...
};
```

Imaginons maintenant que, pour quelque raison que ce soit, l'opération de copie d'un `Rectangle` soit jugée coûteuse (par exemple parce que la masse d'information dans un `Rectangle` se soit accrue au fil du temps). On voudra peut-être alors éviter de copier un `Rectangle` à chaque invocation de `dimension()` puisque cela implique solliciter un constructeur par copie et un destructeur à chaque fois pour la variable temporaire retournée par la méthode.

Une stratégie intuitive serait alors de procéder comme on le fait habituellement dans les langages ne supportant que les sémantiques indirectes (Java et les langages .NET en particulier) et de retourner une référence sur l'attribut dimension_. On aurait alors le code proposé (en abrégé) ci-dessous.

Le problème est que l'appelant de dimension() obtient alors une référence sur un attribut d'un Bouton.

À travers cette référence, l'appelant peut modifier l'état du Bouton à l'insu de ce dernier, *ce qui constitue un bris flagrant (mais trop souvent oublié) d'encapsulation*.

En effet, si l'état du Bouton peut changer à son insu, le Bouton ne peut plus être considéré responsable de sa propre intégrité et, par conséquent, ce Bouton devient vulnérable. Tout appelant hostile pouvant le détruire de l'intérieur; l'intégrité d'un Bouton ne peut plus être garantie.

```
#include "Rectangle.h"
class Bouton {
   // ...
   Rectangle dimension_;
   // ...
public:
   // ...
   Rectangle& dimension() noexcept {
      return dimension_;
   }
   // ...
};
```

Ici, C++ tolérerait qu'on qualifie dimension() de const puisque la méthode ne modifie pas l'état de l'instance propriétaire, mais qualifier cette méthode de const constituerait un bris de constance sur le plan logique (le compilateur C++ ne garantit que la constance bit à bit, ou *Bitwise Constness*, pas la constance logique, ou *Logical Constness*, comme le fait remarquer ***Scott Meyers*** dans [EffCpp]). Soyez éveillés et assurez l'intégrité *logique* de votre classe en apposant const aux endroits opportuns seulement.

Notez que même si la référence sur un Rectangle retournée par la méthode demeurera une référence sur un Rectangle valide *en tant que Rectangle*, il est possible que les politiques de validité d'une dimension de Bouton soient plus strictes que celles assurant la validité d'un Rectangle; priver le Bouton de la capacité d'assurer son intégrité est une faute grave même si certaines règles restent applicables au niveau de Rectangle.

Le suffixe const apposé précédemment à la signature de la méthode dimension() doit être enlevée, à tout le moins pour que l'objet demeure garant de sa propre intégrité.

Dans les langages où les indirections sont les seuls modes d'accès aux objets, tels que Java ou les langages .NET, ces bris d'encapsulation silencieux foisonnent. On les nomme des *Aliasing Bugs*. Soyez alertes, car ces fautes logiques ne peuvent en général pas être rapportées par les compilateurs mais brisent complètement la capacité des objets d'assurer leur intégrité, empêchant pas le fait même la réalisation d'une barrière d'encapsulation correcte.

Heureusement, il y a une solution, simple : retourner une référence constante plutôt qu'une simple référence. Ce faisant, la référence retournée est assujettie aux règles de constance et il devient illégal de lui appliquer une opération ne respectant pas les règles à cet effet.

La capacité de manipuler des objets constants est donc fondamentale :

- un objet doit être en mesure d'utiliser d'autres objets à la fois pour représenter ses états et pour communiquer avec des tiers;
- pour qu'un objet puisse assurer le respect du principe d'encapsulation, il lui faut être certain que ses clients n'obtiendront pas à son insu un accès privilégié à ses entrailles de manière telle qu'il deviendrait possible de le saboter.

```
#include "Rectangle.h"
class Bouton {
   // ...
   Rectangle dimension_;
   // ...
public:
   // ...
   const Rectangle& dimension() const noexcept {
      return dimension_;
   }
   // ...
};
```

Personnellement, je privilégie les retours par valeur. Il est trop facile, surtout en situation de multiprogrammation, de se faire jouer des tours absolument pervers en partageant des indirections entre plusieurs éléments de code client. Le sujet est trop avancé pour le présent document, mais considérez ceci comme une recommandation.

Lorsque les objets sont manipulés par sémantique de valeur, la constance prend une moins grande place conceptuelle (mais demeure utile sur le plan de l'optimisation) du fait que la plupart des opérations susceptibles de compromettre un objet ou l'autre sont aussi susceptibles de générer des copies. En retour, les sémantiques indirectes compromettent souvent, facilement et sournoisement l'encapsulation et, par le fait même, la solidité des programmes.

Vision Java et .NET : immuabilité, ou constantes sur une base classe

Un exemple de code Java couvrant les diverses opérations proposées pour la classe Rectangle serait :

```
class TailleInvalide extends Exception {
   public TailleInvalide() {
      super ("Taille invalide");
   }
}
public class Rectangle {
   private static final short
      LARGEUR_MIN =  1, HAUTEUR_MIN =  1,
      LARGEUR_MAX = 80, HAUTEUR_MAX = 24,
      LARGEUR_DÉFAUT = 5, HAUTEUR_DÉFAUT = 3;
   private static boolean estHauteurValide(short hauteur) {
      return HAUTEUR_MIN <= hauteur && hauteur <= HAUTEUR_MAX;
   }
   private static boolean estLargeurValide(short largeur) {
      return LARGEUR_MIN <= largeur && largeur <= LARGEUR_MAX;
   }
   public Rectangle() {
      setLargeurBrut(LARGEUR_DÉFAUT);
      setHauteurBrut(HAUTEUR_DÉFAUT);
   }
   public Rectangle(short hauteur, short largeur) throws TailleInvalide {
      // versions publiques et blindées
      setHauteur(hauteur);
      setLargeur(largeur);
   }
   protected Rectangle(Rectangle r) {
      setLargeurBrut(r.getLargeur());
      setHauteurBrut(r.getHauteur());
   }
   // accesseurs publics -- interface de consultation
   // accesseurs de premier ordre
   public short getLargeur() {
      return largeur;
   }
   public short getHauteur() {
      return hauteur;
   }
   // accesseurs de second ordre
   public short getPérimètre() {
      return 2 * (getLargeur() + getHauteur());
   }
   public short getSurface() {
      return getLargeur() * getHauteur();
```

```
   }
   // mutateurs publics -- interface de modification contrôlée
   // versions publiques et blindées
   public void setHauteur(short hauteur) throws TailleInvalide {
      if (!estHauteurValide(hauteur)) {
         throw new TailleInvalide();
      }
      setHauteurBrut(hauteur);
   }
   public void setLargeur(short largeur) throws TailleInvalide {
      if (!estLargeurValide(largeur)) {
         throw new TailleInvalide();
      }
      setLargeurBrut(largeur);
   }
   // mutateurs privés -- interface de modification primitive
   // versions privées et rapides
   private void setHauteurBrut(short hauteur) {
      this.hauteur = hauteur;
   }
   private void setLargeurBrut(short largeur) {
      this.largeur = largeur;
   }
   // méthode de projection à la console
   public void dessiner() {
      final int HAUTEUR = getHauteur();
      final int LARGEUR = getLargeur();
      for (short i = 0; i < HAUTEUR; ++i) {
         for (short j = 0; j < LARGEUR; ++j) {
            System.out.print("*");
         }
         System.out.println();
      }
   }
   public Rectangle cloner() {
      return new Rectangle(this);
   }
   //
   // ...plusieurs autres méthodes (laissez-vous aller)...
   //
}
```

Le code C# et le code VB.NET sont du même niveau de complexité, avec les mêmes qualités, les mêmes défauts et les mêmes stratégies de solution. Je m'en tiendrai ici au cas Java par souci d'économie et de simplicité.

Remarquons tout d'abord que la version Java est, pour l'essentiel, fonctionnellement équivalente à la version C++. Cependant, en comparaison avec la version C++, les clauses de constance sont disparues. La raison en est simple : Java et les langages .NET ne supportent pas le concept de constante au niveau des objets.

Autre différence : les classes Java ne sont pas des valeurs, étant manipulées par indirections, ce qui implique le recours au clonage pour la duplication des états.

En apparence, ceci ne semble pas poser de problème. En effet, plusieurs programmes .NET et plusieurs programmes Java existent sur la planète et tout semble fonctionner comme sur des roulettes. En pratique, malgré la réputation de langage dangereux faite à C++ et la réputation de langage sécuritaire faite à Java et aux langages .NET, la situation concrète est que l'absence d'objets constants rend l'immense majorité des programmes Java et .NET vulnérables aux bris d'intégrité et d'encapsulation.

Attention *: Java et les langages .NET n'empêchent pas la rédaction de programmes sécuritaires* mais leurs modèles respectifs compliquent fortement la rédaction de tels programmes. La plupart des programmeurs génèrent du code à risque sans même s'en apercevoir.

Le problème est plus visible si nous reprenons l'illustration à partir de la classe Bouton proposée plus haut. Le code y est en apparence plus simple mais il est, en réalité, beaucoup plus pernicieux. En effet, dans les langages reposant sur des sémantiques indirectes, on ne manipule jamais d'objets; on manipule toujours des indirections vers des objets, typiquement nommées références.

Cela signifie que dans l'exemple à droite, dimension_ n'est pas un Rectangle mais bien une référence sur un Rectangle, et il en va de même pour toute manipulation d'objet.

```
public class Bouton {
   private Rectangle dimension_;
   // ...
   public void setDimension(Rectangle r) {
      // validation, p.ex.: r != null...
      dimension_ = r;
   }
   public Rectangle getDimension() {
      return dimension_;
   }
   // ...
}
class Test {
   public static void main(String [] args) {
      Bouton b = new Bouton();
      Rectangle r = new Rectangle();
      // ...
      b.setDimension(r);
      // ...
      r = b.getDimension();
   }
}
```

Cela implique que l'expression return dimension_ ne retourne pas une copie de dimension_ mais bien une *copie d'une référence sur* *dimension_*. L'appelant obtient donc une référence **directe** sur un attribut du Bouton et peut en modifier l'état à sa guise, sans que le Bouton ne puisse assurer le respect de ses propres politiques de validité. L'implémentation proposée ci-dessus est **dangereuse**.

En Java comme dans les langages .NET, toute opération susceptible de donner une référence directe sur un attribut d'un objet modifiable constitue un bris flagrant d'encapsulation, susceptible d'être lourd de conséquences. Ces bris peuvent résulter de l'obtention (p. ex. : par un accesseur tel getDimension()) ou de l'injection (p. ex. : par un mutateur tel setDimension()) d'une référence.

Le bris d'intégrité va dans les deux sens, affectant autant l'objet que le code client. En effet, dans notre exemple, le code client (la classe `Test`) et son `Bouton` ont tous deux une référence sur un même `Rectangle`, ce qui implique que le `Bouton` peut tout autant modifier le `Rectangle` à l'insu de son client que le client peut modifier le `Rectangle` à l'insu du `Bouton`. **En l'absence de constantes, les deux se mettent mutuellement à risque**.

Deux options existent pour permettre à un objet à la fois d'offrir des services et d'assurer sa propre intégrité : adopter des stratégies de *programmation défensive* et avoir recours à des *classes immuables*.

Stratégies de programmation défensive

L'une des stratégies possibles pour défendre un objet contre les bris d'encapsulation en l'absence de constantes est de faire ce que je nommerai de la *programmation défensive*.

L'idée derrière cette stratégie est de toujours présumer que l'interlocuteur a des intentions hostiles et de dupliquer manuellement le contenu de toute référence transigée avec lui.

Dans cette optique :

- tout accesseur ou toute méthode susceptible d'exposer au monde extérieur une référence sur un attribut doit contenir son propre code de duplication de contenu;
- tout mutateur ou toute méthode susceptible d'accepter en tant qu'attribut une référence transmise par un tiers doit dupliquer le contenu auquel elle réfère avant de s'en servir;
- de même, ne sachant pas ce que fera un objet de ses paramètres, tout client doit dupliquer manuellement ses propres objets avant de les passer en paramètre à une méthode, du moins s'il compte utiliser les objets en question plus tard, et tout client doit dupliquer les objets retournés par une invocation de méthode pour éviter un partage inattendu et compromettant de référence.

```
public class Bouton {
   // ...
   private Rectangle dimension_;
   // ...
   public void setDimension(Rectangle r) {
      // validations ...
      dimension_ = r.cloner();
   }
   public Rectangle getDimension() {
      return dimension_.cloner();
   }
   // ...
}
class Test {
   public static void main(String [] args) {
      Bouton b = new Bouton();
      // ...
      Rectangle r = new Rectangle();
      // ...
      b.setDimension(r.cloner());
      // ...
      r = b.getDimension().cloner();
   }
}
```

Vous remarquerez tout de suite que la programmation défensive est très coûteuse en ressources. Cette stratégie implique d'insérer beaucoup de code manuel de duplication (par clonage, forcément) même si, dans bien des cas, le client et son objet réaliseront la duplication deux fois plutôt qu'une, chacun d'eux devant se protéger et ne pouvant présumer de la bonne foi de l'autre.

Le résultat direct est que le code sera beaucoup plus lent et que la collecte automatique d'ordures sera utilisée beaucoup plus souvent en pratique, le nombre d'objets temporaires (et souvent inutiles) augmentant de manière significative. Ce coût est nécessaire en l'absence d'un mécanisme capable d'assurer la constance des objets dans ces langages.

Entendons-nous : le mot *nécessaire* ici signifie nécessaire *en général*. Pour des systèmes qui ne sont utilisés qu'à l'intérieur d'une entreprise et qui ne sont pas susceptibles d'être exposés au monde extérieur, la compromission de l'encapsulation demeure, mais le risque qu'un tiers n'exploite (volontairement ou accidentellement) cette faille sont plus faibles.

Les bris d'encapsulation sont surtout dangereux lorsque le code est utilisé dans plusieurs milieux ou par plusieurs types de clientèles; pour une `API` ou pour du code de bibliothèque, les fautes de ce genre sont impardonnables.

Détail délicat : le clonage, comme vu dans ***Techniques de clonage*** plus haut, est une opération subjective, qu'on implémente habituellement à l'aide du constructeur par copie protégé de l'objet cloné. Un compilateur ne peut se porter garant du respect par le code de cette façon de faire.

Ainsi, un objet hostile pourrait aisément remplacer `return new Rectangle(this);` par `return this;` dans la méthode `cloner()` de `Rectangle` et toutes les tentatives de duplication subjective du code client seraient en vain.

Immuabilité

La programmation défensive coûte cher et a ses limites. En pratique, faute d'avoir la possibilité d'offrir des garanties de constance, on privilégiera l'immuabilité. En effet, dans les langages `OO` qui ne supportent pas les instances constantes et qui n'offrent qu'une sémantique d'accès indirecte, on privilégiera les classes immuables dans les interfaces des objets.

⇒ Une entité **immuable** n'offre aucun service par lequel il est possible de modifier ses états.

Plusieurs classes standard des infrastructures Java et `.NET` sont immuables, la plus célèbre étant la classe `String` (ou `string` en C#). La raison de ce choix d'immuabilité est la place prépondérante que tient ce type, à cheval entre les rôles de type primitif et de classe à part entière, dans la majorité des langages : étant utilisé partout, si ce type n'était pas immuable, l'encapsulation ne serait, en pratique, réalisée essentiellement nulle part avec ces langages.

Chaque instance d'une classe immuable ne peut plus être modifiée une fois qu'elle a été construite. La programmation à l'aide d'objets immuables sollicite donc à fond les constructeurs, ce qui implique typiquement les mécanismes d'allocation dynamique de mémoire et de collecte automatique d'ordures.

Pour cette raison, la plupart des classes immuables ont une contrepartie qui n'est pas immuable et permet la modification d'états sans imposer un recours démesuré à ces mécanismes.

En Java, la classe `String` a une contrepartie modifiable nommée `StringBuffer`. Sous `.NET`, la classe `string` de C# (un alias pour `system.String`) a une contrepartie modifiable nommée `System.Text.StringBuilder`.

Dans les deux cas, la raison est simple : les manipulations sur des chaînes coûtent *tellement* cher en création et en récupération d'objets temporaires qu'elles constituent des goulots d'étranglement dans un nombre important de programmes et il est essentiel d'offrir une contrepartie efficace aux classes plus typiques mais inefficaces en pratique.

Si nous décidons d'appliquer une stratégie d'immuabilité à la classe `Rectangle`, nous avons deux grandes familles d'options.

La première est de mettre au point une interface de consultation de Rectangle (nommons-la RectangleConsultation) qui n'offre que des services de consultation[105], puis de faire en sorte que Rectangle implémente cette interface.

Ceci implique qu'il faille modifier Bouton pour que cette classe ne retourne que des RectangleConsultation et supprimer son mutateur setDimension().

En effet, l'immuabilité d'une interface n'est qu'apparente, car l'objet derrière l'interface peut offrir les services qu'il veut bien offrir et changer d'état comme bon lui semble; nul ne sait ce qui se cache derrière elle.

Ici, un Bouton malicieux pourrait continuer à modifier le Rectangle derrière un RectangleConsultation donné après avoir offert une référence en apparence immuable sur lui.

```
interface RectangleConsultation {
   short getLargeur();
   short getHauteur();
   short dessiner();
}
class Rectangle
   implements RectangleConsultation {
   // ...
}
class Bouton {
   Rectangle dimension_;
   // ...
   public RectangleConsultation getDimension() {
      return dimension_;
   }
}
```

L'option d'offrir une interface immuable sur un objet qui peut ne pas l'être permet[106] à l'objet (ici, une instance de Bouton) de permettre la consultation des états de ses attributs sans toutefois permettre leur modification impromptue par le code client. Selon ce modèle, Bouton sera typiquement immuable, ou encore offrira des services tels des mutateurs qui délégueront les demandes aux attributs après validation seulement.

Une classe comme Bouton n'offrira pas de service tel setDimension(Rectangle) ou setDimension(RectangleConsultation) puisque ces services permettraient à du code client malicieux de lui injecter un cheval de Troie.

[105] Prenez *services de consultation* au sens de *services qui ne modifient pas l'objet*.

[106] … dans la mesure où le type caché derrière l'interface est inaccessible au code client, sinon rien ne l'empêchera d'obtenir par transtypage une référence qui n'est pas immuable derrière l'interface immuable et de tout gâcher. Ici comme ailleurs, on privilégiera les types privés aux types publics ou protégés.

La seconde est de définir deux classes à part entière, dont on ne peut dériver[107] et qui se connaissent mutuellement comme le font les classes Java `String` et `StringBuffer`.

Truc : interdire d'hériter de classes immuables permet d'éviter qu'un tiers ne conçoive un enfant de l'une d'elles dans le but d'infiltrer le système et de le saboter. Pour les interfaces, évidemment, ceci n'est pas une solution.

Cet idiome est habituellement celui à privilégier pour prévenir les bris d'encapsulation dans les langages à sémantique d'accès indirecte et sans soutien aux objets constants.

Faire en sorte qu'une instance de l'une de ces deux classes puisse être construite à partir d'une instance de l'autre et inversement permet ensuite de générer un seul niveau de copie défensive, à même l'objet. Un `Bouton` manipule à l'interne un `RectangleModifiable` pour fins de performance mais n'en expose, sur demande, qu'un duplicata immuable (un `Rectangle`).

De manière peut-être surprenante, le recours à des constructeurs de copie publics dans ces classes est indiqué ici du fait que ces classes sont terminales (elles ne peuvent avoir de dérivés, étant qualifiées `final`). Il n'y a pas de risques de *Slicing* comme ceux examinés dans la section sur les techniques de clonage, plus haut.

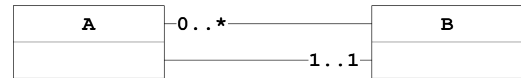
```
final class RectangleModifiable {
   //
   // ... Rectangle, plus haut ...
   //
   public RectangleModifiable(Rectangle r) {
      // ...
   }
}
final class Rectangle {
   // ... Rectangle, plus haut,
   // mais immuable...
   public Rectangle(RectangleModifiable r) {
      // ...
   }
}
class Bouton {
   private RectangleModifiable dimension_;
   // ...
   public Rectangle getDimension() {
      return new Rectangle(dimension_);
   }
}
```

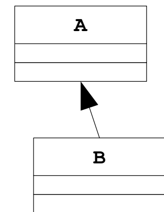
Cet idiome n'est malheureusement pas une panacée. Il ne permet pas d'introduire de constance une instance à la fois ou d'offrir des garanties de constance méthode par méthode. Il exige aussi un effort de programmation supplémentaire considérable en imposant la duplication d'une partie importante du code.

[107] On parle donc de classes `final` en Java, `sealed` en C#, et `NotInheritable` en `VB.NET`.

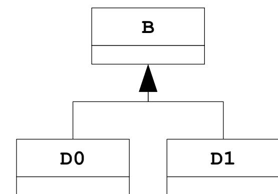
Annexe 00 – Résumé de la notation UML abordée dans ce document

Cardinalité. Dans le schéma à droite, toutes instance de la classe A est en relation avec une et une seule instance de la classe B, et toute instance de la classe B est en relation avec un nombre arbitrairement grand (et pouvant être nul) d'instances de A.

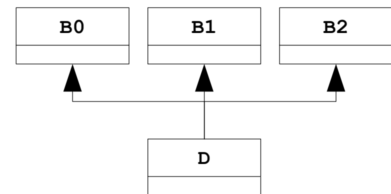




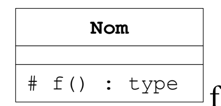
Héritage simple. Dans le schéma à droite, la classe B dérive (hérite) de la classe A.



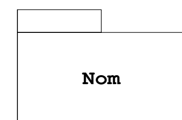
Héritage simple (*deux classes sœurs*). Dans le schéma à droite, les classes D0 et D1 sont sœurs parce qu'elles ont en commun un parent nommé B.



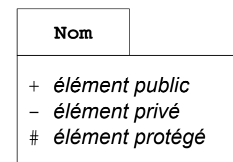
Héritage multiple. Dans le schéma à droite, la classe D a trois parents distincts nommés B0, B1 et B2.



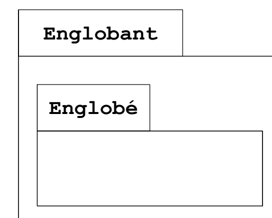
Membre protégé. Dans le schéma à droite, la classe Nom possède une méthode protégée f() de type type.



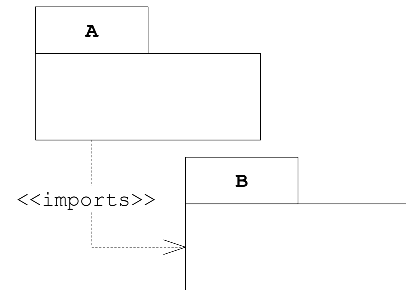
Regroupement. Le schéma à droite présente le regroupement (paquetage, espace nommé, assemblage, module, *etc.*) nommé Nom.



Regroupement (*suite*). Le schéma à droite présente le regroupement (paquetage, espace nommé, assemblage, module, etc.) nommé Nom et contenant des éléments.



Regroupement (*suite*). Le schéma à droite présente le regroupement nommé Englobant contenant un regroupement nommé Englobé.



Regroupement (*suite*). Le schéma à droite présente le regroupement nommé A dépendant du regroupement nommé B de par une relation d'importation.

Annexe 01 – Discussions sur quelques réflexions choisies

Cette annexe présente quelques discussions portant sur des questions de réflexion parsemées ici et là dans le document. Ces questions ont été choisies (et posées) délibérément parce que les réponses ne sont pas aussi évidentes et banales qu'il n'y paraît à première vue.

Réflexion 01.0 : le polymorphisme est subjectif

Une question de réflexion, soulevée dans la section sur le polymorphisme et les indirections, porte sur la question du polymorphisme et des méthodes de classe ou d'instance. Le polymorphisme peut-il s'appliquer sur une méthode de classe?

> Vous avez peut-être remarqué que `X::Yo()` et `Y::Yo()` auraient, en temps normal, pu être des méthodes de classe puisqu'elles n'accèdent, pour compléter leur tâche, à aucun membre d'instance. Pourtant, ici, ce sont des méthodes d'instance. À votre avis, quelle est la raison derrière cette décision technique?

La réponse brève est non. Qui dit action subjective dit aussi sujet : le polymorphisme est une sélection dynamique de la version la plus appropriée d'une méthode virtuelle en fonction du type effectif d'une instance.

Pour invoquer `x.f()` ou `y->f()` de manière polymorphique, il faut :

- que la méthode `f()` soit virtuelle; et
- que `x` soit une référence ou que y soit un pointeur (selon le cas).

Remarquez la syntaxe : `x` et `y` sont des indirections vers des *instances*. Une invocation de méthode de classe, comme `T::f()` pour la classe `T`, ne laisse aucune place à la détermination dynamique du type réellement impliqué : c'est `T`. Dans le cas de `x` et de `y`, par contre, le type effectif est au moins le type dont `x` est une référence ou au moins le type vers lequel pointe `y`.

Sur le plan technique, réaliser une invocation polymorphique de manière efficace demande de passer par une table dans laquelle sont entreposées les adresses des méthodes à invoquer pour un objet donné. Chaque instance possédant au moins une méthode polymorphique possède aussi une telle table, par définition (conséquence : les objets polymorphiques sont un peu plus gros qu'il n'y paraît), et chaque invocation polymorphique implique une indirection à travers une table de pointeurs (ce qui est plus lent qu'un appel direct de fonction, soit, mais demeure optimal si le dynamisme associé au polymorphisme est requis).

Cette table (la `vtbl`, dans le jargon C++) est entreposée dans chaque instance polymorphique. C'est ce qui permet, à peu de frais, de déterminer, pour tout objet polymorphique, la bonne méthode pour une invocation polymorphique donnée. Elle est donc associée à `this`; or, il n'y a pas de `this` dans une méthode de classe.

Conséquence : seules les méthodes d'instance peuvent être polymorphiques. Sans sujet, point de subjectivité, point d'action subjective.

Réflexion 01.1 : annoncer les exceptions

Pourquoi la fonction `f()` n'annonce-t-elle pas `throw(TailleIllegale)`, et ne l'annoncerait pas même si les spécifications d'exceptions n'étaient pas dépréciées (comme elles le sont depuis C++ 11)? Cela mérite réflexion.

Devant la fonction `f()` ci-dessous, nous avons posé la question de réflexion visible à droite.

```
class TailleIllegale {};
void f(int n) {
   if (n <= 0) throw TailleIllegale{};
   X *p = new X[n];
   g (p, n);
   delete[] p;
}
```

La question soulevée est : devrions-nous apposer la mention `throw(TailleIllegale)` à la signature de la fonction `f()`? Et la réponse est un retentissant **non**.

Ici, `f()` peut lever `TailleIllegale`… entre autres choses. Remarquez que la fonction `f()` instancie `n` instances par défaut de `X`, puis invoque `g()`. Il nous faudrait, pour garantir que `f()` ne risque de soulever que `TailleIllegale`, être certains *a priori* que :

- le constructeur par défaut de `X` ne soulèvera pas d'exception (quoique ceci puisse être fait si `X::X()` est `noexcept`), outre bien sûr `TailleIllegale`;
- la fonction `g()` ne lèvera jamais d'exception (outre `TailleIllegale`) elle non plus, ce qui peut aussi s'avérer possible si `g()` expose la mention `noexcept` ou la mention `throw(TailleIllegale)`; et
- avoir la certitude *a priori* que l'opérateur `new[]` ne lèvera jamais d'exception, ce qui est impossible pour deux raisons : en premier lieu, `new[]` peut lever `std::bad_alloc`, et en second lieu, `new[]` peut être surchargé de multiples manières échappant à `f()` (nous y reviendrons dans d'autres volumes).

Ici, la seule chose raisonnable qu'on puisse faire pour les qualifications d'exceptions de `f()` telle qu'elle est écrite est de ne pas en indiquer, ce qui est un honnête aveu d'impuissance : nous ne pouvons offrir de garanties à son sujet, tout simplement.

Réflexion 01.2 : exceptions sans effets secondaires

> Le message contenu dans une exception standard C++ est un `const char *`, pas une `std::string`. Est-ce un bon choix?.

Tel qu'indiqué dans la section ***Exceptions et vie des objets***, une exception ne devrait jamais, dans ses méthodes, lever elle-même une exception, car cela pourrait entraîner les programmes qui s'en servent dans une désastreuse spirale de dégradation.

La question de réflexion `01.2` demandait, à cet effet, si le choix d'utiliser une chaîne `ASCIIZ` brute (un `const char *`) pour représenter un message dans les exceptions standards de C++ (constructeur d'`exception`, méthode polymorphique `what()`) est un choix judicieux. Après tout, la bibliothèque standard offre une classe `std::string`, n'est-ce pas?

En fait, le choix d'utiliser un type primitif est le seul choix raisonnable pour représenter un message dans une exception :

- utiliser une `std::string` impliquerait, à l'interne, réaliser de l'allocation dynamique de mémoire, ce qui peut lever `std::bad_alloc`; et
- utiliser une `std::string` comme type retourné par la méthode `what()` aurait les mêmes conséquences. Toute invocation de `what()` pourrait lever une exception.

Les copies de types primitifs (incluant des références et des pointeurs qui ne résultent pas d'une allocation dynamique de mémoire) ou les copies d'objets sans effets secondaires (sans allocation dynamique de ressources) comme les classes vides ou la classe `HorsBornes` de la section ***Exceptions et vie des objets*** sont des opérations qui ne lèveront jamais d'exceptions. Conséquemment, ces types se prêtent mieux à la représentation d'exceptions ou d'états d'exceptions que ne le font les types plus complexes comme la classe `std::string`.

Annexe 02 – Covariance et contravariance (bref)

La **covariance** est un concept qu'on retrouve dans plusieurs langages. Comme vu dans la section ***Covariance : spécialisation des types des méthodes polymorphiques***, C++ support les types de retour covariants pour les méthodes virtuelles. Java et C# supportent pour leur part la covariance pour les tableaux, au sens où si un `Y` peut y être traité comme un `X`, alors un `Y[]` peut être traité comme un `X[]`… C'est une erreur que les concepteurs des deux langages reconnaissent aujourd'hui mais avec laquelle il leur faut composer pour supporter le code existant. C++ permet de supporter explicitement ce type de covariance par programmation générique [POOv02] mais n'offre pas ce support par défaut.

La **contravariance** est un concept connexe à la covariance. L'explication est un peu technique, mais en C++, la contravariance découle du fait qu'il est possible de capturer un pointeur sur un membre d'un parent dans un pointeur sur un membre d'un enfant, du fait que cette conversion « inversée » ne peut mener à aucune conséquence néfaste.

Par exemple :

```
class B {
   // ...
public:
   virtual void service_important(int);
   virtual ~B() = default;
};
class D : public B {
   // ...
public:
   void service_specialise(int);
};
int main() {
   using pmethB = void (B::*)(int);
   using pmethD = void (D::*)(int);
   pmethB pmb0 = &B::service_important; // Ok
   // pmethB pmb1 = &D::service_specialise; // illégal
   pmethD pmd0 = &B::service_important; // Ok
   pmethD pmd1 = &D::service_important; // Ok
}
```

Intension et extension

Les concepts de covariance et de contravariance sont cousins de ceux d'**intension** et d'**extension** que l'on retrouve dans la littérature (merci à **Florian Bœuf-Terru** pour cette piste) :

- l'**extension d'une classe** est l'ensemble de ses instances. Les instances d'une classe sont aussi des instances de ses ancêtres, de manière transitive[108]. Conséquemment, l'extension d'une classe est un sous-ensemble de l'extension de ses ancêtres;
- **les extensions sont covariantes**, au sens où elles varient en suivant la spécialisation dans une hiérarchie de classes;
- l'**intension d'une classe** est l'ensemble de ses propriétés[109]. Une classe dérivée hérite les propriétés de ses parents et en vient à enrichir cet ensemble en y ajoutant les siennes. Ainsi, les intensions du parent forment un sous-ensemble des intensions de chacun de ses enfants;
- pour cette raison, **les intensions sont contravariantes**, évoluant dans le sens inverse de celui de la spécialisation.

[108] Je dois avouer ne pas savoir à quel point ce concept accommode les héritages autres que publics.

[109] Prenez ce terme au sens large, pas dans une acception technique comme celle mise de l'avant par les langages C# ou VB.NET.

Individus

Les individus suivants sont mentionnés dans le présent document. Vous trouverez, en suivant les liens proposés à droite du nom de chacun, des compléments d'information à leur sujet et des suggestions de lectures complémentaires. Avis aux curieuses et aux curieux!

Florian Bœuf-Terru	Diplômé de la cohorte 06 du Diplôme de développement du jeu vidéo (DDJV) offert à l'Université de Sherbrooke
James Gosling	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#james_gosling
Andrew Hunt	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#andrew_hunt
Barbara Liskov	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#barbara_liskov
Scott Meyers	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#scott_meyers
Pierre Prud'homme	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#pierre_prudhomme
Dennis Ritchie	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#dennis_ritchie
Bjarne Stroustrup	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#bjarne_stroustrup
Herb Sutter	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#herb_sutter
Dave Thomas	http://h-deb.clg.qc.ca/Sujets/Orthogonal/Individus-importants.html#dave_thomas
David R. Tribble	Je n'ai pas d'informations particulières à son sujet au moment d'écrire ceci.

Références

Les références qui suivent respectent un format quelque peu informel. Elles vous mèneront soit à des notes de cours de votre humble serviteur, soit à des documents pour lesquels mes remarques sont proposées de manière électronique et à partir desquels vous pourrez accéder aux textes d'origine ou à des compléments d'information.

[BooExcS] http://www.boost.org/community/exception_safety.html

[CppStdLib] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#cpp-standard-library

[EffCpp] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#effective-cpp

[ExcCpp] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#exceptional-cpp

[FwkDes] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#framework_design_guidelines

[GosIntv] http://www.artima.com/intv/gosling34.html

[HMult] http://www.parashift.com/c++-faq-lite/multiple-inheritance.html

[hdEBCO] http://h-deb.clg.qc.ca/Liens/Optimisation--Liens.html#ebco

[hdEnPar] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Enchainement-parents.html

[hdPol] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Programmation-par-politiques.html

[hdSelPar] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Selecteur-classe-parent.html

[hdTBN] http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Truc-Barton-Nackman.html

[ItfHarm] http://blog.cleancoder.com/uncle-bob/2015/01/08/InterfaceConsideredHarmful.html

[ISOCast] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1993/N0349a.pdf

[JavaAcCtl] http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html

[JavaAlias] http://www.ibm.com/developerworks/java/library/j-jtp06243/index.html

[LiskovSubs] http://en.wikipedia.org/wiki/Liskov_substitution_principle

[MExcCpp] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#more-exceptional-cpp

[POOv00] POO – Volume 00, par Patrice Roy et Pierre Prud'homme.

[POOv02] POO – Volume 02, par Patrice Roy et Pierre Prud'homme.

[POOv03] POO – Volume 03, par Patrice Roy et Pierre Prud'homme.

[PragProg] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#pragmatic-programmer

[StrouDE] http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#design_evolution_cpp

[SutterNvi] http://www.gotw.ca/publications/mill18.htm

[UnsStack] http://www.awprofessional.com/content/images/020163371x/supplements/Exception_Handling_Article.html