

PG208

Programmation Orientée Objet / Langage C++

TP5

Y. Bornat A. Valade

February 17, 2025

1 Classes abstraites

Il peut arriver qu'il soit nécessaire que plusieurs classes aient un héritage commun, mais que l'utilisation de la super-classe n'ait aucun sens, voir pose problème dans certains cas. Il est alors possible de créer une classe dont le seul intérêt est de servir à la création de sous-classes. Une telle classe, dite *abstraite* ne doit pas permettre d'être instanciée. Par contre, elle propose tout un nombre de facilités pour créer des sous-classes.

Pour concevoir une telle classe, il suffit qu'il y ait au moins une méthode virtuelle initialisée à 0 comme dans l'exemple ci-dessous:

```
1  class AbstractClass {
2      ...
3      virtual <typename> reality(...) = 0;
4      ...
5  };
```

La méthode `reality` est dite *virtuelle pure*, au sens où il n'est pas possible de définir la fonction `AbstractClass::reality1`. Pour que la sous-classe d'une classe virtuelle puisse être utilisée, il suffit de définir toutes les méthodes virtuelles pures de sa super-classe.

Il est interdit d'utiliser une classe abstraite en tant que :

- type de déclaration
- type de paramètre de fonction
- type de retour de fonction
- type de conversion

Pour résumer, vous pouvez uniquement déclarer des sous-classes ou utiliser des références ou pointeurs vers une classe abstraite.

Testez sur l'exemple précédent en déclarant la méthode `Animal::bouge` comme abstraite. Dans ce cas, pour représenter l'Anémone, il faut créer une autre sous-classe pour les invertébrés.

Exemples d'usages :

- Définir un ensemble de méthodes communes à des objets qui *a priori* n'ont rien d'autre en commun. On peut imaginer une classe définissant un véhicule immatriculé. Cette classe définit l'ensemble des méthodes et attributs liés aux services d'immatriculation, mais aucune description du véhicule lui-même n'est fournie car cela restreindrait cette description. Ici, on cherche à rendre une portion de code la plus générique (et réutilisable) possible.
- Laisser une autre partie du programme implémenter certains aspects essentiels au fonctionnement de la classe. Telle quelle est fournie, une telle classe n'est pas pleinement fonctionnelle et ne doit donc pas être utilisée.

- Interdire l'accès à certains paramètres : essentiellement si le fonctionnement de la super-classe n'est pas stable¹, cette technique dissuade un développeur d'utiliser des éléments non perennes.

2 Membres finaux / Classes finales

Il est aussi possible d'interdire de sous-classer une méthode ou une classe complète. Pour cela, il suffit de préciser le mot-clé `final` à la fin de la déclaration.

```

1  class Toto final {
2      ...
3      ...
4  };
5
6  class Tata {
7      void methodTata (...) final;
8      ...
9  };

```

Dans cet exemple, il n'est pas possible de créer une sous-classe de `Toto`. Il est possible de créer une sous-classe de `Tata`, mais il ne sera pas possible de surcharger la méthode `methodTata`.

3 Sémantiques

En programmation objet, on oppose souvent les sémantiques d'entité et de valeur. On va juste les définir, histoire qui vous ne soyez pas perdus dans une documentation.

La sémantique de valeur signifie que votre classe est surtout utile pour coder une valeur. C'est le cas de la classe `Polynomial`. Si deux objets de cette classe sont égaux, ils sont interchangeable, et il n'y a aucune raison pour utiliser l'un plutôt que l'autre. La seule chose qui compte au final, c'est leur valeur.

La sémantique d'entité va au contraire considérer que si deux objets ont la même valeur, c'est un (mal)heureux hasard, mais les deux entités restent bien distinctes d'un point de vue de la gestion des données du programme. Un bon exemple est de considérer une classe *mamifère*. Si, pour deux entités différentes l'attribut `domestique` est à `True`, la chaîne `alimentation` vaut `"Croquettes"`, la chaîne `logement` vaut `"panier"`, ... Il suffit qu'une de ces entités soit stockée dans une variable qui s'appelle `chien` et l'autre dans une variable qui s'appelle `chat` pour deviner qu'il n'est pas pertinent de les fonctionner. Bien souvent, les opérateurs mathématiques n'ont aucun sens pour des classes à sémantique d'entité².

4 Exceptions

LA plupart des erreurs produites par le programme

A partir du moment où il existe un mécanisme permettant de faire remonter des erreurs, il est possible de l'utiliser de façon à faire remonter les comportements exceptionnels, inattendus ou pour lesquels le code n'a pas été prévu.

Le mot-clé pour générer un tel événement est `throw`. Deux grosses différences avec les assertions : il est inconditionnel (pas grave, on peut le placer dans un `if`), et il prend un argument (comme `return`). Le type (la classe) de l'argument va coder le type d'exception. Il existe de nombreux types, et chaque mise à jour de la norme en apporte de nouveaux. Pour C++11, certains des plus basiques sont : `std::bad_typeid`, `std::bad_cast`, `std::bad_weak_ptr`, `std::bad_function_call`, `std::bad_alloc`, `std::bad_array_new_length`. Attention, `std::bad_array_new_length` est une forme particulière de `std::bad_alloc`.

Il existe également des types plus élaborés : `std::invalid_argument`, `std::domain_error`, `std::length_error`, `std::out_of_range` et `std::future_error` qui sont des formes particulières de `std::logic_error`. Mais aussi `std::range_error`, `std::overflow_error`, `std::underflow_error`, `std::regex_error` et `std::system_error` qui sont des formes particulières de `std::runtime_error`.

¹la stabilité d'une classe n'a pas la même définition que celle d'un programme. Un programme instable plante souvent, mais une classe ou une interface instable signifie que ses attributs peuvent disparaître, changer de type, ou de signification à la moindre mise à jour.

²Les esprits pourront toujours rappeler l'existence de l'équation, `ennuis = chat + chien`, mais elle est un peu (ca)pillotracée

Les types élaborés ont la particularité d'accepter une chaîne de caractère lors de leur création. Il est ainsi possible d'écrire un vrai message d'explication du problème. Ils sont tous définis par `#include <stdexcept>`.

Histoire de vous faire la main, écrivez un programme tout simple, qui écrit du texte dans le terminal, lance une exception avec un message de votre choix, puis écrit une autre ligne de texte.

Évidemment, la dernière ligne de texte n'apparaît pas. Par contre, vous aurez remarqué que le texte de l'exception s'affiche sans référence de ligne ou de fichier. En effet, comme une exception est supposée avoir des causes externes, ce n'est pas le programme qui est en cause. Le texte de description doit donc se cantonner à décrire le problème, et éventuellement ses causes supposées.

4.1 Récupération

Les exceptions ne sont pas que des assertions un peu modifiées. Elles ont une particularité bien plus intéressante : on peut les intercepter. Par exemple, on peut imaginer qu'une fonction `foo` appelle une sous-fonction `bar` qui a pour mission de charger des données dans un fichier et de renvoyer un résultat. Si l'accès au fichier échoue, `bar` ne peut pas renvoyer de résultat alors qu'elle y est obligée. Elle déclenche donc une exception. Si `foo` n'a rien prévu de particulier, l'exécution du programme s'arrête et l'exception est affichée comme une erreur. Mais si `foo` est préparée à l'éventuel déclenchement d'une exception, elle peut la rattraper, demander de spécifier un autre nom de fichier à l'utilisateur et retenter d'exécuter `bar`.

4.1.1 syntaxe

Pour générer une exception, on utilisera le code suivant (en une ligne):

```
1 throw std::runtime_error("On insulte joyeusement l'utilisateur si mauvais");
```

mais si vous préférez quelque chose de plus verbeux, on peut décomposer les étapes de la création :

```
1 std::string text {"Cher Utilisateur\n"};  
2 text += "t'es nul !";  
3 std::runtime_error exc {text};  
4 throw exc;
```

Vous êtes bien sûr libre de concevoir des messages de taille quelconque. L'essentiel étant de bien identifier le problème.

Pour intercepter l'exception, la syntaxe est la suivante (à placer dans une fonction) :

```
1 // any code before ...  
2 try {  
3     // possibly faulty code  
4 } catch (std::range_error& ex) {  
5     // exception management  
6     ex.what(); // retrieve the exception message  
7     throw ex;  
8 } catch (std::runtime_error& ex) {  
9     // another possible exception  
10     std::cout << ex.what() << std::endl; // print the exception message  
11 } (... ) {  
12     // all other exceptions  
13     // if this happens, it is generally bad news  
14     throw;  
15 }  
16 // any code after ...
```

Le bloc suivant `try` des lignes 2 à 4 est de longueur quelconque. Si la moindre exception se déclenche à l'intérieur de ce bloc, il est interrompu et l'exécution passe les différents types possibles.

Selon le type de l'interruption, le bloc correspondant est exécuté. Notez que `std::range_error` doit être testé *avant* `std::runtime_error`. Comme le premier est un cas particulier du second, les deux correspondent à un `std::runtime_error`. Si le test était fait dans l'ordre inverse, le second ne serait jamais utilisé car les exceptions auraient déjà été traitées par le premier.

Redéclencher ou pas l'exception avec `throw` ne dépend que de la volonté du concepteur et de ses contraintes.

4.1.2 Pourquoi c'est intéressant

Les exceptions énumérées ci dessus sont celles qui peuvent être indifféremment interceptées ou libérées. Par contre, il est possible de passer n'importe quel type ou classe avec `throw`. Par contre, il devient impératif d'intercepter l'interruption. Si ce n'est pas le cas, on obtient alors une erreur au même titre qu'une assertion. Il sera impossible d'accéder à la valeur renvoyée, le message affichera uniquement son type.

On se retrouve donc avec un canal de retour de valeur indépendant de la valeur renvoyée par une fonction, et surtout non typé. Il est possible de renvoyer n'importe quel type ou classe à condition qu'il y ait une clause `catch` qui puisse la récupérer. On peut ainsi utiliser ce genre de code :

```
1  try {
2      subfunc();
3  } catch (int& i) {
4      std::cout << "got an int   = " << i << std::endl;
5  } catch (double& d) {
6      std::cout << "got a double = " << d << std::endl;
7  } catch (const std::string& s) {
8      std::cout << "got a string = " << s << std::endl;
9  } (...) {
10     std::cout << "Ouch ! got a real exception" << std::endl;
11     throw;
12 }
```

Par contre, il ne faut pas perdre de vue que les exception, comme leur nom l'indique, doivent rester exceptionnelles car elles sont (beaucoup) moins performantes que des retours de fonction classiques. Leur rôle doit rester cantonné aux situations qui sortent du cadre normal de fonctionnement et pour lesquels il n'est pas rentable de concevoir un canal de retour mieux structuré.

4.2 Référencement

Il est possible (conseillé) de préciser dans le prototype d'une fonction si elle est susceptible de déclencher des exception, et, si oui de quel type. Une fonction dont on est sur qu'elle ne déclenchera jamais d'exception sera suivie du mot-clé `noexcept`. Par exemple :

```
1  int fact(int n) noexcept;
```

Au contraire, pour une fonction qui peut renvoyer une exception de type `int`, `double` ou `std::system_error`, on écrira :

```
1  int fact(int n) throw(int, double, std::system_error);
```

Ces ajouts de définition offrent deux avantages :

- L'utilisateur de la fonction sait s'il doit gérer des exceptions
- Le compilateur peut se permettre certaines optimisations quand il sait si une fonction ne produit pas d'exception.

Par définition, un destructeur de classe est toujours `noexcept` car le C++ ne peut gérer qu'une seule exception à la fois.

4.3 Limites

Les exception ne sont pas utilisables dans tous les cas. Notamment, leur comportement et leur syntaxe sont très différents dans les constructeurs. Dans le cadre de cet enseignement nous éviterons simplement de déclencher des exception depuis les constructeurs.

De plus, pour les exceptions comme pour les assertions, si le programme se termine, les destructeurs des variables ne sont pas appelés. Cela peut parfois poser problème.

4.4 Exercice

Pour cet exercice, l'objectif est de créer une class `ifstream_exc` ou `ofstream_exc` (au choix) qui permet de lire ou écrire un fichier. L'objectif est de lancer une exception si l'ouverture du fichier ne s'est pas bien passée. (Tout le monde a certainement le souvenir d'avoir, au moins une fois, galéré sur un `segfault`

donc la cause était l'utilisation d'un `FILE *` dont l'initialisation avait échoué, non ?). En C++, c'est pire, l'écriture ou la lecture ne se font pas, mais il n'y a pas de système de signalisation. On a juste les méthodes `good` et `fail` qui renvoient un booléen sur l'état du flux.

Comme il n'est *pas possible* de jeter des exceptions dans le constructeur, vous pouvez (au choix) ne pas ouvrir le fichier dans le constructeur, mais forcer l'appel à `open` (propre, mais oldfashion) ou provoquer l'exception dans l'opérateur de flux `<<` ou `>>`.

L'intérêt est bien sur de fournir au moins un programme qui intercepte l'exception et qui transforme un gros crash en petit message.

5 Polymorphisme

La notion de polymorphisme est liée au fait de n'écrire qu'une seule fois du code qui s'applique à plusieurs types de données différents.

Nous avons vu la version utilisant les `template`. Cette version n'est qu'un artifice de syntaxe. Dans la pratique, si vous utilisez une telle fonction sur trois types différents, le compilateur créera trois fonctions différentes, chacune étant appelée dans le contexte qui lui est propre. Le code est donc plus court, mais pas le programme fourni. C'est tout de même une avancée pour éviter la duplication de code.

Voici une version plus poussée. En effet, si une fonction accepte une certaine classe en entrée, elle acceptera toutes ses sous-classes. Techniquement, si la fonction utilise un passage par valeur, l'argument sera dépouillé de tous les membres constituant sa sous-classe mais si elle utilise un passage par référence, l'argument gardera sa classe et les membres d'origine. Ce comportement est possible en langage compilé grâce au fait que les références ont une taille unique (celle d'un pointeur), quel que soit le type de l'objet auquel elles font référence. Cette fois, le code de la fonction est unique, mais comme il faut déréférencer les arguments à chaque fois, il y a une légère perte de performances. Cependant, le gain (en temps de développement, en fiabilité du code et en complexité des concepts manipulables) est tellement important sur de nombreux autres aspects, que cela représente un investissement rentable, même sur le plan des performances finales.

La version la plus extrême du polymorphisme n'est pas possible en C++ car c'est un langage fortement typé. Mais vous l'avez déjà utilisé extensivement en *Python*. En effet, pour les langages faiblement typés, il n'est pas nécessaire de définir ou connaître le type de données manipulé avant l'exécution du programme. Cela permet de panacher les types de contenu des conteneurs, l'écriture de fonctions polymorphes sans `template` ni équivalent, ou encore le fait qu'une variable puisse stocker successivement un flottant, une chaîne de caractères puis un entier. Ce comportement est techniquement faisable en C++, il suffirait de créer une classe `Nawak` capable de contenir, au choix, n'importe quel type de données, et de surcharger les opérateurs courants. La capacité du compilateur à appeler le constructeur idoine fera le reste. Il y aura cependant des contreparties :

- il ne sera pas possible de passer naturellement de la classe `Nawak` à un type ou une classe de base.
- il faut réécrire (ou surcharger) toutes les fonctions/bibliothèques qui utilisent des types de base en entrée (à cause du point précédent)
- La moindre opération sur un entier nécessitera de tester le type de données, les performances globales seront nettement en retrait. Ce n'est pas l'esprit du C ou du C++.

6 Conclusion

6.1 Ce que nous n'avons pas vu

Ce cours est nécessairement restreint, parce que le nombre d'heures est limité, mais aussi parce que nous ne sommes pas en filière info. Vous avez par contre les bases pour vous en sortir efficacement sans aide complémentaire. Cette partie est là juste pour que vous connaissiez l'existence de certains principes.

6.1.1 Héritage multiple

Il est possible d'écrire une classe qui soit une sous-classe de deux super-classes différentes. On parle alors d'héritage multiple. Ça ne pose aucun problème. Si des attributs ont le même nom, on les distingue grâce à leur espace de nom (`A::name`) et `B::name` ou `A` et `B` sont les deux super-classes). Pour les méthodes, on les utilise comme des fonctions libres, en ajoutant un pointeur vers l'instance de la sous-classe comme premier argument : `A::method(&subcl, autres args...)`.

Là où ça devient tordu, c'est que ces deux super-classes peuvent aussi être des sous-classes, et qu'il y a potentiellement plusieurs fois la même classe. Par exemple, un carré est à la fois rectangle particulier, et un losange particulier. Mais les deux sont des parallélogrammes particuliers, ici, la classe grand-mère est la même, et l'instance grand-mère est la même aussi. Par contre, un prêt bancaire peut être à la fois une dépense et un revenu. Les deux sont associés à une banque... mais pas forcément la même. Dans ce deuxième cas, la classe grand-mère est la même, mais pas les instances... bref ! c'est possible, mais renseignez vous.

6.1.2 Les itérateurs

Pour faire une boucle `for` en utilisant la syntaxe avancée³.

```
1 for (float elem : tab)
```

On utilise une classe un peu spéciale qui s'appelle un itérateur. Dans sa forme la plus simple, un itérateur se comporte comme un pointeur nu. Si `tab` est un vecteur ou un tableau, le code précédent est équivalent à :

```
1 for (float *elem = &(tab[0]); elem != &(tab[tab.size()-1]); elem++) {  
2     ...  
3 }
```

Bien sûr, dans ce code, on n'utilise pas `elem` mais `*elem` pour accéder à l'élément de `tab`. Mais l'esprit reste le même. L'avantage d'un itérateur, c'est qu'il est possible de surcharger l'opérateur de déréférencement (`*`), l'incréméntation, la comparaison Au final, on obtient toute la souplesse de l'arithmétique des pointeurs, mais sur des conteneurs quelconques, même si ce sont des structures chaînées.

6.1.3 Les foncteurs

Vous avez remarqué qu'il est possible de surcharger l'opérateur `()` pour utiliser une instance de classe comme une fonction. Ce principe est utilisé de manière assez générique en C++. En effet, une telle instance peut être vue comme une fonction, avec ou sans paramètres, stockée dans une variable (l'instance). Cela s'appelle un *foncteur*. Il est techniquement possible de concevoir une fonction/méthode qui reçoit un foncteur comme argument, et l'applique à tout ou partie des éléments d'un conteneur. Ça vous étonne si on vous dit que la bibliothèque standard en est pleine ?

Allez faire un tour du côté de la documentation de `<algorithm>`, vous y trouverez du plus basique (`for_each` décrit il y a deux lignes) au plus complexe : fonction de tri qui prend un itérateur (donc conteneur quelconque, potentiellement custom) et un foncteur pour la relation d'ordre (donc fonction d'ordre absolument quelconque).

6.2 Et les autres langages ?

Pour les aspects de programmation orienté objet, nous n'avons fait qu'ouvrir la boîte, mais les possibilités des langages objet ont peu de limites. Voici quelques subtilités qu'on trouve dans d'autres langages ...

6.2.1 Propriétés (*property* au sens Python)

Les propriétés au sens Python consiste en une paire de fonctions qui permettent de récupérer (*getter*) et définir (*setter*) une valeur que n'a pas nécessairement de variable associée. Cette paire de fonction sont appelées par le même nom, et sans parenthèses. Par exemple, pour une classe `c`, et une propriété `p`,

³Dans d'autres langages, ce genre de structure s'appelle une boucle *foreach*. Cette appellation infuse un peut dans la terminologie du C++, même si le mot-clé reste `for`

le code `c.p = c.p * 2` appelle la fonction de récupération (malgré l'absence de parenthèses), multiplie la valeur renvoyée par deux et appelle la fonction de définition avec le résultat du calcul (malgré l'absence de parenthèses). Ce mécanisme, absent du C++, a plusieurs avantages :

- Il est possible de conserver la même interface utilisateur, même si la structure interne de la classe est radicalement différente, et qu'un attribut public a été supprimé.
- L'utilisateur peut manipuler des valeurs comme des attributs, même s'ils ne sont pas directement accessibles (besoin de réserver l'accès en mode multi-thread, ressource présente dans un fichier, sur le réseau, ...)
- On peut tester les valeurs écrites dans les attributs pour vérifier leur validité

6.2.2 classes variables (*Meta-classes* en Python)

Vous aviez du mal avec les foncteurs (fonction stockée dans une variable et manipulable) ? Avec certains langages avancés, il y a pire ! Il est possible de stocker un type de donnée (une classe) dans une variable, et de le modifier, avant d'en créer une ou plusieurs instances. L'intérêt est que la classe est créée pendant l'exécution, au besoin, puis des variables contenant une instance de cette classe sont créées. Même pas peur. En Python, on appelle cela une méta-classe. Mais le mot-clé est `type`. Ainsi, une classe est en fait une instance d'une classe dont le nom est `type`. Pas complètement idiot quand on y pense.

Un cas d'usage serait, par exemple, dans le cadre de la taxonomie des espèces. On peut y retrouver les relations d'héritages de classe, avec une classe mère *être-vivant*, des classes filles *vegetal* ou *animal*, des classes petite-fille *vertébré*, ... Le problème est qu'il existe plusieurs centaines de catégories... Pourquoi ne pas stocker l'arbre dans un fichier, et construire les classes automatiquement ? En python, on peut. En C++, le plus abordable est de faire le programme qui écrit les fichiers `.cpp` et `.hpp` pour ensuite recompiler. Il y a plus élégant, d'autant qu'il n'est pas possible de mettre à jour les classes sans tout recompiler.

6.3 Pour finir

Vous n'êtes pas des spécialistes en POO, mais vous avez les bases pour savoir si, pour un projet donné, vous en avez besoin, ou pas. Pour savoir ce que cela peut vous apporter, ou pas. Et l'effort qu'il faudra faire pour en exploiter toute la puissance. Sur le long terme, utiliser une approche objet est rentable. La vraie difficulté est de savoir si votre projet va atteindre le seuil qui rend l'objet rentable. Il n'y a pas d'autres solution que de faire appel à l'expérience, en espérant rester objectif.