PG208
Programmation Orientée Objet / Langage C++
TP4

Y. Bornat A. Valade

March 11, 2025

La séance précédente nous a permis de voire les classes et leur principe. Cette séance sera dédiée à plusieurs petites choses bien pratiques. Que ce soit pour mieux gérer un code complexe, ou pour exploiter la force des concepts qu'elles contiennent. En effet, les classes ne sont pas *que* des types de données très évolués, on peut les utiliser dans des conditions bien différentes.

1 Programmation dite *par contrat*

1.1 Principe

Cette partie est un enrobage un peu théorique pour un concept que vous manipulez depuis un moment sans vous en rendre compte. Quand vous programmez, vous posez en permanence des invariants que vous vous obligez à respecter, et qui vous permettent de garantir que votre code fonctionne. C'est par exemple le cas pour une boucle qui parcours les éléments d'un vecteur avec un indice. L'invariant de boucle (l'indice désigne une case du tableau), pose une contrainte sur l'indice, il doit être compris entre 0 et la taille du tableau -1.

Dans les classes, c'est la même chose. Les différents attributs sont cohérents entre eux, mais rien ne le garanti s'ils sont affectés au hasard. Il faut aussi que les arguments des constructeurs permettent de garder la cohérence voulue. Utiliser une classe signifie aussi faire tout ce qu'il faut pour que ses données soient cohérente. On appelle ça la *programmation par contrat*. Par exemple, pour la classe `Polynomial` le contrat est que, `m_coeffs` contienne au moins une valeur et que, s'il y a plusieurs valeurs dans `m_coeff`, celle d'indice le plus élevé soit non nulle. (sinon, le degré du polynôme n'est pas lié au nombre de coefficients).

Pour gérer tout ça, le C++ a hérité du C une commande de vérification de condition rapide : les assertions.

1.2 Syntaxe

Pour utiliser les assertions, il faut d'abord inclure la bibliothèque correspondante : `#include <cassert>`. Ensuite, il suffit d'appeler la *macro*[1] `assert()`. Le seul argument qu'elle prend est une condition. Si la condition est vraie, rien ne se passe. C'est que tout va bien. Si elle est fausse, c'est qu'un contrat n'est pas respecté, et le programme plante méchamment.

Le langage ne permet pas d'ajouter de message explicatif, mais l'erreur donne le numéro de ligne ou s'est passé l'assertion négative et, bien souvent, recopie la ligne fautive (avec la condition donc). La technique souvent employée est donc d'ajouter la chaine de caractère à la condition comme suit :

```
1    assert(resp==42 && "La vie, l'univers, ....");
```

Techniquement le programme fait un `ET` bouléen entre `resp==42` et la chaine de caractère. Étant donné qu'une chaîne de caractères renvoie toujours `true`, il n'y a que le test qui est réellement pris en compte.

L'usage des commentaires reste possible, mais il n'est pas pratiqué car rien ne garanti que les commentaires de la ligne seront recopiés dans le terminal.

[1] Si vous ne savez pas ce qu'est une macro, vous survivrez. On dira alors que c'est *presque* comme une fonction

1.3 Intérêt

Cette technique présente de nombreux avantages :

- Elle est rapide
- Le lecteur peut rapidement identifier qu'on vérifie la validité de quelque chose.
- Elle est désactivale sans effort. En effet, le comportement de `assert` est défini en fonction d'une définition du pré-compilateur. Il suffit alors d'ajouter `#define NDEBUG` avant l'inclusion de `<cassert>`, et tous les tests seront ignorés.

Là ou le dernier point est intéressant, c'est que les définitions peuvent être faites dans la ligne de commande du compilateur (option `-D` pour *gcc/g++*). Ainsi, en compilant avec `-DNDEBUG`, aucun test de type `assert` ne sera fait, et on gagne en performances.

Mieux encore, dans le cas d'une compilation multifichier, il est possible d'activer cette option (ou pas) fichier par fichier.

1.4 Les tests unitaires

La notion de *test unitaire* est liée à la vérification des composants de base d'une bibliothèque. Le principe est de faire de nombreuses assertions élémentaires qui vérifient les contrats. Un développement saint multiplie les tests unitaires de l'ensemble des structures utilisée avant de les assembler dans un système complet. La vérification à l'échelle du projet est constituée des tests *d'intégration*, qui ne font pas partie des objectifs de ce cours. Encore une fois, si on effectue des `assert` pour les tests unitaires, ils peuvent être dans le code final, et simplement retirés grâce à la définition de `NDEBUG`. Le principe est de ne jamais retirer un test unitaire qui a été plac" dans le code pour éviter les régressions lors d'une mise à jour.

Par contre, les assertions sont faites pour détecter les anomalies dans le code (debugage d'une bibliothèque ou vérification de la validité d'arguments). elles ne *doivent pas* être utilisées pour les problèmes liés à l'environnement (erreur de saisie utilisateur, indisponibilité d'un fichier, coupure réseau, ...). Pour cela, il y a les exceptions !

2 Exceptions

A partir du moment ou 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 évènement 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 type, 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`.

Les types élaborés ont la particularité d'accepter une chaine 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'apparait 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 cantoner à décrire le problème, et eventuellement ses causes supposées.

2.1 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'éxécuter `bar`

2.1.1 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 sur libre de concevoir des message 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(); // retreive 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'interieur de ce bloc, il est interrompu et l'exécution passe les différents types possibles.

Selon le type de l'interruption, le bloc correspondant est éxécuté. 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 exception auraient déjà été traitées par le premier.

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

2.1.2 Pourquoi c'est intéressant

Les exceptions énumérées ci dessus sont celles qui peuvent être indifféremment interceptées ou libérées. Il est possible de passer n'importe quel type ou classe avec `throw`, mais 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();
```

```
    } catch (int& i) {
        std::cout << "got an int   = " << i << std::endl;
    } catch (double& d) {
        std::cout << "got a double = " << d << std::endl;
    } catch (const std::string& s) {
        std::cout << "got a string = " << s << std::endl;
    } (...) {
        std::cout << "Ouch ! got a real exception" << std::endl;
        throw;
    }
```

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 cantoné 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.

2.2 Référencement

Il est possible (conseillé) de préciser si un fonction peu, ou pas, déclencher une interruption. Par défaut, toute fonction *peut* en déclencher. Par contre, une fonction dont on est sur qu'elle ne déclenchera jamais d'exception sera suivie du mot-clé `noexcept`. Par exemple :

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

Cette précision permet au compilateur de faire quelques optimisations.

Par définition, un destructeur de classe est toujours `noexcept` car il y a de fortes chances qu'il soit exécuté lors d'une exception, et le C++ ne peut gérer qu'une seule exception à la fois.

2.3 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.

2.4 Exercice

Pour cet exercice, l'ojectif est de créer une classe `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.

3 Sous-classes et héritage

Le grand intérêt de la programmation orientée objet ne réside pas (que) dans la possibilité de créer des types nouveaux manipulables comme les types de base. Si nous reprenons l'exemple des polynômes, il existe des types particuliers de polynômes qui pourraient bénéficier de méthodes ou attributs supplémentaires. Par exemple, en cryptogtaphie, on modélise des clés de chiffrement par des polynômes dont les coefficients sont 1 ou 0, il deviendrait alors intéressant de fournir des méthodes spécifiques à cette application..

Pour cela, il est possible de faire une sorte de grosse surcharge de classe, et d'ajouter tous les éléments nécessaires à un usage nouveau ou spécifique, en réutilisant les parties déjà écrites de la

classe existante. La nouvelle classe sera alors une *sous-classe*, et la classe existante sa *super-classe*[2]. L'intérêt est que la sous-classe garde tous les attributs de sa super-classe, il n'y a donc pas besoin de tout redéfinir. On dit que la sous-classe *hérite* des attributs de sa super-classe. D'où la terminologie *Mère-Fille*

La dérivation d'une classe ne peut pas se faire n'importe comment. Telle qu'elle est définie, la classe `Polynomial` ne peut pas être sous-classée pour s'étendre aux exposants négatifs et faire de la transformée en z par exemple. En effet, une sous-classe est un cas particulier de sa super-classe, pas une extension de cette dernière. Si quelque chose ne peut pas être décrit par une classe, il ne doit pas non plus être définissable par une de ses sous-classes. Si *Foo* est une sous-classe de *Bar*, alors toute entité de *Foo* est aussi une entité de *Bar*.

4 Syntaxe

4.1 Déclaration

Supposons que nous voulons faire une bibliothèque qui décrit des éléments géomériques, il y aura certainement une classe *rectangle*, dont une sous-classe serait *carré*.

Nous aurions alors la déclaration suivante :

```
1 class Rectangle {
2     Rectangle(...);
3     ...
4 };
5
6 class Square : public Rectangle {
7     Square(...);
8     ...
9 }
```

Le mot-clé `Public` de la ligne 6 signifie que tous les éléments de `Rectangle` seront visibles des utilisateurs. En utilisant le mot-clé `private` à la place, aucun de ces éléments ne sera visible. L'utilisateur ne verra donc pas que le carré est aussi un rectangle l'intérêt est limité :).

Une sous-classe est considérée comme *utilisatrice* de sa super-classe elle n'a donc pas accès à ses éléments privés. Ce comportement est relativement logique. Si ce n'était pas le cas, il suffirait de créer une sous-classe pour outrepasser le caractère privé de certains membres.

4.2 Membres protégés

Il est possible d'utiliser le mot-clé `Protected` pour signifier qu'un membre n'est pas accessible aux utilisateurs, mais qu'il l'est pour ses sous-classes. Le code ressemble alors à :

```
1 class Rectangle {
2     public:
3         Rectangle(...);
4         ...
5     protected:
6         ...
7     private:
8         ...
9 };
```

Le statut `Protected`, doit être restreint au strict minimum pour ne pas inciter d'usages non désirés, voire dangereux.

4.3 Construction

En l'état, notre classe `Square` est peu utilisable, car elle n'a pas de constructeur. Ou plutôt, nous n'avons pas vu comment écrire son constructeur. En effet, sans savoir si le constructeur de `Rectangle` initialise des attributs privés, il est obligatoire d'y faire appel dans le doute. Pour forcer le programmeur à faire cet appel, le constructeur de `Square` doit être défini en mentionant explicitement le constructeur de la super-classe.

[2] selon les terminologies, on parle egalement classe mère et classe fille, ou classe de base et classe dérivée, peut-être pour épargner la susceptibilité de ces pauvres sous-classes qui se sentent dévalorisées.

```
Square::Square(....) : Rectangle(...) {
    ....
}
```

Avec cette syntaxe, on remarque que le constructeur de `Rectangle` ne peut recevoir que des arguments construits simplement à partir de ceux du constructeur de `Square`. C'est une restriction. Il reste bien sur la possibilité d'utiliser un constructeur de `Rectangle` qui ne fait rien et d'initialiser les attributs hérités à l'aide d'autres méthodes, mais ce mécanisme doit être prévu par la super-classe.

Concernant le destructeur, chacun s'occupe de ses propres affaires. Autrement dit, si vous écrivez un destructeur pour une sous-classe, il n'y a pas besoin d'appeler le destructeur de la super-classe. Le compilateur fait ça tout seul. Le destructeur de la super-classe est appelé après la fin du destructeur de la sous-classe. Cela signifie également que le destructeur de la sous classe ne doit *jamais* libérer les ressources de sa super-classe.

5 Comportement

La sous-classe (`Square` dans notre cas) est une classe à part entière. Il faut simplement se rappeler que ses membres incluent aussi ceux de sa super-classe. On dit que `Square` *hérite* des membres de `Rectangle`.

Ce type de fonctionnement est très pratique quand il est nécessaire de créer plusieurs classes qui ont des éléments en commun. Ces élement communs sont donc définis dans une super-classe. Toutes les sous-classes n'auront alors plus besoin de s'en préoccuper.

5.1 Échauffement

Écrivez `TRINOM`, une sous-classe de `Polynomial`, mais dont le degré vaut 2 ou moins. Techniquement, cela consiste en une déclaration, deux constructeurs et une surcharge.

Pour le fun, vous pouvez maintenant ajouter la méthode `roots`, qui utilise le discriminant pour calculer les racines du trinôme. Pour cela, vous aurez besoin de la racine carrée (`sqrt`) disponible dans `<cmath>`. Bonne nouvelle, contrairement au C, la bibliothèque mathématique est accessible sans option de compilation.

5.2 Une limite des langages compilés

Dans notre exemple de géométrie, toute instance de `Square` est également une instance de `Rectangle`. Il est donc possible de passer une instance de `Square` à une fonction `foo` qui attend un `Rectangle`. Il n'y aura pas de conversion[3]. Par contre, seuls les membres de `Rectangle` seront accessibles à l'intérieur de `foo`. Même s'il ont été surchargés par `Square`. Ce type de comportement est restrictif mais normal, le compilateur ne peut pas anticiper que `foo` a en fait été appelée sur une sous-classe au lieu de la classe attendue.

À moins que ...

6 Méthodes virtuelles

Les méthodes virtuelles répondent précisement à ce problème. Si une méthode est délcarée comme *virtuelle*, l'association entre l'instance et sa méthode n'est plus faite au moment de la compilation, mais au moment de l'éxécution. Le choix du code à éxécuter dépend donc de la classe réelle de l'instance, pas de la classe attendue.

La syntaxe est la suivante pour une méthode `whoami` :

```
class Rectangle {
    Rectangle(...);
    virtual void whoami();
    ...
};

class Square : public Rectangle {
```

[3] Ce n'est donc pas le même comportement que quand on donne un `int` à une fonction qui attend un `double`. Dans ce dernier cas, la valeur est convertie.

```
    Square(...);
    void whoami() override;
    ...
}
```

Notez bien que la méthode doit avoir le préfixe `virtual` dans la super-classe et le suffixe `override` dans la sous-classe. Mentionner ce statut n'est nécéssaire que pour la déclaration. La définition des méthodes est normale.

Petit détail : pour fonctionner, ce principe a besoin de toujours s'appliquer à une instance de la sous-classe. Mais si une fonction fait une copie d'une instance de sous-classe alors qu'elle attend la super-classe, la copie sera du type de la super-classe. Par exemple, la fonction suivante posera problème :

```
void foo(Rectangle obj) {
    std::cout << obj.whoami() << std::endl;
}
```

Quelle que soit l'instance passée à la fonction, la méthode appelée sera toujours celle de `Rectangle`. Cela vient du fait que la fonction effectue un passage par valeur. Une instance `Square` sera alors copiée comme une instance `Rectangle`. Pour obtenir le comportement attendu, il faut que la fonction récupère l'original, donc une référence.

7 Un peu de pratique ...

Assez de discours. Pour bien comprendre les choses, il faut les manipuler. A partir du deuxième point, éxécutez votre code à chaque étape, et comparez leur résultat. Pour simplifier les choses, les attributs seront publics.

- Écrivez une classe `Animal`, cette classe doit contenir :
  - un attribut `pattes` de type `int`.
  - un attribut `cri` de type `std::string`.
  - un constructeur qui reçoit le nombre de pattes et le cri en argument
  - une methode `talk` qui affiche le cri
  - une méthode virtuelle `bouge` qui génère une exception (ou affiche un texte d'erreur, selon votre choix, si vous ne voulez pas faire de try/catch)
- Écrivez le main qui crée `casper` une instance de `Animal`, et appelle `talk` et `bouge` sur cette instance.
- Écrivez une fonction `marche`, qui reçoit un `Animal` et appelle `bouge` autant de fois qu'il y a de pattes. Ajoutez un appel à `marche(casper)` dans le main.
- Ajoutez une variable `anemone` de type `Animal` (l'anémone a un seul pied, et ne dit rien, m'enfin si elle dit quelquechose, on ne l'entend pas). Appelez `talk`, et `marche`.
- Écrivez `Vertebre` qui est une sous-classe de `Animal`, elle doit contenir
  - un attribut `nb_os`.
  - un constructeur qui reçoit la valeur des trois attributs en argument et fait référence au constructeur de `Animal`.
  - une méthode `bouge` déclarée `override` et qui affiche "lève la pate", suivit de "pose la patte".
  - Rien d'autre
- Dans le main, créez `chat`, une instance de `vertebre`, ajoutez un appel à `chat.talk()`, et à `chat.marche()`
- modiifez la fonction `marche` pour qu'elle reçoive une référence. et relancez le test.
- Comment se débrouiller pour que la fonction `marche` puisse accéder à `nb_os` ???

8 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:

```
class AbstractClass {
    ...
    virtual <typename> reality(...) = 0;
    ...
};
```

La méthode `reality` est dite *virtuelle pure*, au sens ou il n'est pas possible de définir la fonction `AbstractClass::realityl`. 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 declarer 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. Ce n'est pas demandé car ça ne présente que peu d'intéret.

9 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.

```
class Toto final {
    ...
    ...
};

class Tata {
    void methodTata (...) final;
    ...
};
```

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`.

10 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 ditinctes 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 funsionner. Bien souvent, les opérateurs mathématiques n'ont aucun sens pour des classes à sémantique d'entité[4].

11 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 connaitre le type de données manipulé avant l'éxécution 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 chaine 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++.

12 Conclusion

12.1 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'existance de certains principes.

[4]Les esprits pourront toujours rappeler l'existance de l'équation, `ennuis = chat + chien`, mais elle est un peu (ca)pillotractée

12.1.1 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 ou 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.

12.1.2 Les itérateurs

Pour faire une boucle `for` en utilisant la syntaxe avancée[5].

```
for (float elemt : 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 :

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

Bien sûr, dans ce code, on n'utilise pas `elemt` mais `*elemt` 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'incrémentation, 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.

12.1.3 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/methode 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).

12.2 Et les autres langages ?

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

12.2.1 Propriétés (*property* au sens Python)

Les propriétés au sens Python consiste en une paire de fonctions qui peremettent de récuperer (*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`,

[5] Dans 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 plublic 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

12.2.2 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'éxécution, 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 idot 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.

12.3 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 vrai 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.