PG208
Programmation Orientée Objet / Langage C++
TP5

Y. Bornat A. Valade

February 13, 2025

Pour cette séance, nous allons commencer tout doux, sur la documentation et l'utilisation des classes. Dans un second temps nous verrons comment il est possible de spécialiser des classes déjà existantes.

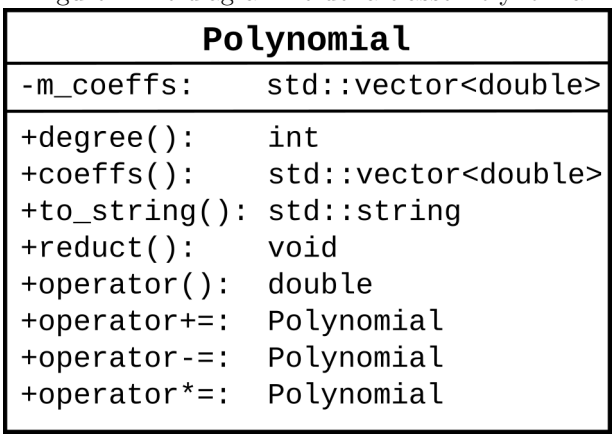
1 Diagramme UML

Les classes sont des types de données relativement complexes à gérer. Et encore, nous verrons plus tard que l'organisation de classes dans un projet peut vraiment devenir un casse-tête. A ce niveau, la documentation est un enjeu majeur, car personne ne peut gérer de grand projet seul, mais une documentation mal construite est contre-productive. Pour y remédier, un système de représentation standardisée a été adopté[1]. Il s'agit de la représentation *UML* (Unified Modelling Language). Ce standard propose, entre autres, une représentation visuelle des différents éléments d'un système, tels que les classes, les objets, les relations entre eux, les interactions et les comportements. UML s'applique également à la description de machines d'états, d'organisation (architecture système ou organisation sociale), *etc*. Nous nous limiterons à une approche rapide limitée à son application en *POO*.

Les classes sont représentées sous forme de rectangles divisés en trois parties : le nom de la classe, les attributs et les méthodes. Quand il y a une relation entre deux classes, on trace un trait ou une flèche entre elles. Pour le détail, c'est une suite de règles assez simples :

- Le nom de la classe est dans le compartiment du haut. Il est centré, écrit en gras (quand c'est possible) et respecte la casse (Majuscules/minuscules).
- Les attributs sont écrits dans le deuxième compartiment. Chaque attribut est écrit sur une ligne séparée, avec sa visibilité, son nom et son type. Les deux derniers sont séparés par le caractère *deux-points*. La visibilité peut être publique (+), privée (-) ou protégée (#).
- Les méthodes arrivent en dernier. Chaque méthode est écrite sur une ligne séparée, sa visibilité, son nom avec ses paramètres entre parenthèses et type de retour. Le caractère : sépare la parenthèse fermante et le type de retour.
- lorsque le type d'un attribut de la classe est une autre classe[2] on trace une flèche entre la classe source (qui référence) et la classe cible (qui est référencée).
- si deux classes ont une référence croisée (chacune des deux a une référence sur l'autre), ce n'est pas une flèche mais un trait.
- pour une classe source, on indique le nombre de référence à côté de la base de la flèche (ou du trait si c'est une référence croisée)
- il est possible de décrire la relation avec deux ou trois mots max le long des traits.
- la disposition, les couleurs, *etc* sont laissés à l'appréciation artistique du concepteur.

Figure 1: Le diagramme de la classe Polynomial



Par exemple, pour la classe `Polynomial`, on retrouvera le diagramme de la figure 1:

Il se *pourrait* que, dans un avenir proche, vous ayez un rapport ou un gros compte-rendu à faire concernant un code en programmation orientéee objet. La communication avec des diagrammes UML étant systématique sur les projets en *POO*, entrainez vous sur les codes que vous allez écrire prochainement.

2 Fonction *amies*

2.1 Definition

Une fonction *amie* d'une classe est définie comme une fonction libre, mais elle a accès à l'ensemble des membres de la classe, même ceux qui sont notés privés. Il faut donc bien retenir que cette fonction, bien que non membre, fait partie intégrante de la classe.

Dans le cas de la classe `Polynomial`, on la déclare comme à la ligne 13 du code suivant :

```
1   class Polynomial {
2       public:
3           Polynomial(std::vector<double> coeffs);
4           Polynomial(double c);
5           Polynomial(double b, double c);
6           Polynomial(double a, double b, double c);
7           ~Polynomial();
8
9           int degree();
10      private:
11          std::vector<double> m_coeffs;
12
13      friend std::ostream & operator<<(std::ostream & os, Polynomial & p);
14  };
```

2.2 Utilisation

L'intérêt de cette catégorie de fonctions est de pouvoir surcharger des fonctions ou opérateurs de façon transparente, ou, au contraire, de créer une fonction qui puisse facilement être surchargée. En effet, la surcharge ne doit pas être vue comme une façon d'occulter l'existant, mais de le compléter et de l'enrichir. De la même façon, il est possible de poser les bases d'un système par quelques classes, mais de laisser l'utilisateur l'enrichir grâce aux surcharges. Il est alors essentiel de laisser à l'utilisateur la liberté de faire evoluer les ressources.

Les fonctions *amies* ne sont pas techniquement indispensables. Par exemple, pour `Polynomial`, vous avez écrit des fonctions libres qui n'étaient pas amies. Mais ces fonctions ont dû faire appel à la

[1]Notez bien qu'il s'agit d'un standard, pas d'une norme. Mais ce dernier est assez bien suivi.

[2]Techniquement il assez rare qu'une classe soit stockée dans une autre, on utilise beaucoup plus souvent des références.

méthode `.coeffs()`, ce qui les rend plus lentes à l'éxécution et consomme des ressources mémoire. Ce surcoût est le prix de la standardisation des accès, et ne pose pas trop de problème dans le cas général. Mais il devient gênant si la fonction libre est fournie en même temps que la classe.

2.3 Opérateurs : méthodes ou fonctions amies ?

2.3.1 Oui ! on a le choix

La très grosse majorité des opérateurs *unaires* peuvent être déclarés, au choix, comme une fonction membre[3] ou comme une fonction libre. La fonction membre ne prend pas d'argument, la fonction libre prend un argument dont le type est la classe (ou une référence vers la classe)

De même les opérateurs *binaires*, peuvent être déclarés comme fonction membre qui ne nécessite qu'un seul argument (le deuxième terme), ou une fonction libre qui reçoit deux arguments (les deux termes). Si le programme est cohérent, au moins un des deux termes sera la classe que nous sommes en train d'écrire.

Mais si les fonctions *amies* permettent d'accéder aux membres privés, comment faire le choix entre fonction libre ou fonction membre ?

2.3.2 La différence

Elle réside dans la permissivité que vous autorisez à l'usage. Si vous écrivez une fonction membre, vous imposez que le premier terme soit rigoureusement de la classe que vous avez créée. Si vous écrivez une fonction libre, vous permettez au compilateur de s'adapter à la situation grâce aux constructeurs disponibles.

Par exemple, si vous surchargez l'opérateur `+` dans une fonction membre de `Polynomial`, vous serez capable de calculer `p1+p2` où `p1` et `p2` sont des `Polynomial`, mais aussi `p1+a` où `a` est un `double` puisqu'il existe un constructeur à partir d'un `double`. Comme il existe un constructeur pour un `double` à partir d'un `int`, vous pouvez aussi calculer `p+i`. Par contre, il sera impossible de calculer `a*p`, parce que le premier terme doit nécessairement être un `Polynomial`.

Si l'opérateur `+` est surchargé dans une fonction libre, le compilateur sera capable de comprendre que pour `a+p`, il faut d'abord appeler le constructeur pour convertir `a` en `Polynomial`, puis appeler l'opérateur `+`. C'est généralement le comportement recherché car il conserve la commutativité de l'addition.

2.3.3 Usage

En C++, l'usage pour la surcharge d'un opérateur est de définir l'opérateur de modification (`+=`, `*=`, ...) comme fonction membre qui met à jour l'instance. Puis de définir l'opérateur binaire comme fonction libre qui crée une copie du premier terme, et appelle l'opérateur de modification sur la copie. Cela apporte toute la souplesse évoquée ci-dessus, sans avoir besoin de définir trop de fonctions amies, ni d'avoir de code redondant. Cela a un coût en termes de performances, mais les compilateurs modernes reconnaissent la situation et minimisent (voire annulent) les pertes.

2.3.4 Exercice

- Re-écrivez les fonctions libre de la classe `Polynomial` en tant que fonctions amies pour améliorer les performances.
- Si ce n'est pas déjà fait, assurez vous qu'il est possible de créer un polynôme quelconque en utilisant uniquement des opérations de base sur le polynome identité que vous noterez `X`.

3 Fonctions/méthodes constantes

Pour rappel, lors du passage des instances comme arguments ou résultat, le C++ effectue des copies par defaut. Il est parfois intéressant d'utiliser des références pour obtenir l'original, que ce soit par souci de performances ou par nécessité de le modifier. Dans le premier cas, il est est possible de

[3] on ne parlera pas de méthode ici car on ne les appelle pas en mentionnant le nom de l'instance suivi du nom de la fonction séparés d'un point

récupérer une référence constante : on travaille bien sur l'original, mais le compilateur interdit de le modifier. C'est par exemple le cas pour la surcharge de l'opérateur `<<`. Mais le compilateur refuse un `const Polynomial &`. C'est parce qu'on fait appel à la méthode `degree`. En effet, le compilateur n'est pas capable de vérifier si, oui ou non, cette méthode modifie son instance. Résultat : dans le doute, il refuse.

La solution consiste à préciser que la méthode `degree` ne modifie pas son instance. Cela se fait grâce au mot-clé `const` placé à la fin de la déclaration (`int degree() const;`), et lors de la définition de la méthode (`int degree() const { ... }`). Ce principe s'applique à la fois aux méthodes et aux fonctions membres, c'est pour cela qu'on utilise l'appellation générique *fonction constante*.

Par contre, ce n'est pas de la magie. En effet, si dans une fonction constante, on tente de modifier l'instance ou on fait appel à une fonction non constante qui utilise une référence, la compilation échouera.

4 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*[4]. 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*.

5 Syntaxe

5.1 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 :).

[4] 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.

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.

5.2 Mombres 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 :

```
class Rectangle {
    public:
        Rectangle(...);
        ...
    protected:
        ...
    private:
        ...
};
```

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

5.3 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.

```
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.

6 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.

6.1 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 la méthode du 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.

6.2 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[5]. 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 ...

7 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 {
    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.

8 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

[5] 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.

  - 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` ???

9 Conclusion

A ce stade, vous disposez de toutes les cartes pour faire de la programmation objet en C++ a un niveau très correct. Il ne manque qu'un peu de pratique. nous verrons cela lors de la prochaine séance et du projet.