Université de Nice-Sophia Antipolis
Polytech ELEC4

Mardi 3 décembre 2019
Durée: 2h

C++ Héritage
Travaux Dirigés – Séance n. 7

---

1 Introduction

La notion de *réutilisabilité* est une caractéristique des logiciels qui exprime la possibilité de réutiliser tout ou partie des composants qui les forment. Nous reviendrons sur cette notion au second semestre lorsque nous parlerons de façon détaillée de la méthodologie de conception objet.

Dans ce TD, nous allons évoquer un outil fondamental des langages à objets, *l'héritage*, ainsi que deux autres notions étroitement liées, mais toutes aussi fondamentales, le *polymorphisme* et la *liaison dynamique*.

2 Héritage

Nous voulons représenter des figures géométriques. Commençons par les rectangles. Pour les représenter, nous écrirons, par exemple, la classe `Rectangle` suivante :

```
class Rectangle {
/* Invariant : largeur>=0, longueur>=0, */
private:
  double largeur;
  double longueur;

public:
  Rectangle(const double l=0, const double L=0) :
    largeur(l), longueur(L) {}

  double getLargeur() const;
  double getLongueur() const;

  void setLargeur(const double l);
  void setLongueur(const double L);

  double perimetre() const;
  double surface() const;
};
```

exercice 1) Dans le fichier `Rectangle.cpp`, écrivez l'implémentation des méthodes précédentes et dans un fichier `main.cpp`, écrivez la fonction `main` qui teste votre classe `Rectangle` avec :

```
Rectangle r1(2.6, 4.42), r2;

std::cout << r1.perimetre() << std::endl;
r1.setLargeur(3.9);
```

exercice 2) On souhaite maintenant représenter des carrés. Écrivez la classe `Carré` sur le modèle de la classe précédente. Vous aurez une variable membre `cote` et les méthodes `getCote` et `setCote`.

Dans la méthode `main`, ajoutez et testez :

```
Carre c1(3.8), c2;

std::cout << c1.perimetre() << std::endl;
c2.setCote(2.9);
```

En comparant les classes `Carre` et `Rectangle`, on peut remarquer qu'elles se ressemblent énormément. On peut alors se demander s'il est possible de *réutiliser* la classe `Rectangle` pour définir la classe `Carre`. La réponse est *oui* grâce à la notion d'*héritage* qui permet de définir une classe `B`, appelée classe *dérivée*, par spécialisation/extension d'une autre classe `A`, appelée *superclasse* (ou classe *mère*, ou encore classe *de base*).

Pourquoi est-il naturel de faire hériter la classe `Carre` de `Rectangle` ? Tout simplement parce qu'un carré *est-un* rectangle *particulier* (la largeur est égale à la longueur). Une classe dérivée est donc une *spécialisation* de sa superclasse. Toutefois, elle peut aussi être vue comme une *extension* dans le mesure où on pourra ajouter de nouveaux attributs et méthodes dans la classe dérivée.

D'une façon générale, à chaque fois que la relation *est-un* peut être appliquée entre deux classes, la relation d'héritage devra être utilisée.

La classe dérivée `B` est héritière au sens qu'elle possède tous les attributs et les méthodes de sa superclasse `A` en plus de ses propres attributs et méthodes. Toutefois, elle n'hérite pas des constructeurs, du destructeur, du constructeur de copie, ni de l'opérateur d'affectation.

L'en-tête de la déclaration une classe `B` qui dérive d'une classe `A`, sera :

```
class B :  mode dérivation A
```

où *mode dérivation* est soit **private** (comportement par défaut), **protected** ou **public**.

Nous avons déjà vu **private** et **public** , le langage C++ propose un troisième accès **protected**, qui limite l'accès des attributs ou des méthodes aux classes héritières.

L'héritage est contrôlé par le *mode dérivation*. S'il est **private**, tous les attributs et les méthodes **public** ou **protected** de la superclasse deviennent privés dans la classe dérivée. Si le mode est `proctected`, ils deviennent protégés. Enfin, si le mode est **public**, tous les attributs et les méthodes de la superclasse gardent leur mode d'accès dans la classe dérivée. Ce dernier mode est le mode de dérivation habituel.

La classe dérivée est donc amenée, si nécessaire, à définir ses propres constructeurs, destructeur, constructeur de copie, et opérateur de copie. Lorsqu'elle définit son constructeur, par défaut le constructeur par défaut de la superclasse est d'*abord* exécuté. Mais, le constructeur de la classe dérivée peut également faire appel à un constructeur particulier de la superclasse avec la notation :

```
B(...) :  A(...) {}
```

Le constructeur de la classe `A` sera exécuté *avant* celui de `B`.

exercice 3) Récrivez la classe `Carre` en la faisant hériter de la classe `Rectangle` selon le mode **public**. Dans la classe `Rectangle`, vous déclarerez les variables `largeur` et `longueur` **protected**. Dans la classe `Carre`, vous définirez le constructeur `Carre(double c)` qui initialise le coté du carré. Compilez et exécutez à nouveau votre fonction `main`. Constatez que dans l'énoncé `c1.perimetre()`, la méthode appliquée à un objet de type `Carre` est obtenue par héritage de la classe `Rectangle`.

Lorsque un objet de la classe dérivée est détruit, le destructeur de cette classe dérivée s'applique *d'abord*, puis celui de la superclasse.

exercice 4) Mettez évidence ce comportement, en définissant un destructeur dans `Rectangle` et un autre dans `Carre`.

---

exercice 5) Est-il possible d'appliquer la méthode `setLongueur` à un objet de type `Carre` ? Est-ce

normal et est-ce correct ? Comment éviter cela ? Modifiez votre programme en conséquence ?

3 Redéfinition de méthodes

exercice 6) Dans le fichier `Rectangle.cpp`, donnez l'implémentation de la méthode :

```
std::string quiSuisJe() const;
```

pour qu'elle renvoie la chaîne de caractères *"un rectangle"*.

exercice 7) Dans votre fonction `main`, appliquez la méthode `quiSuisJe` sur `r1` et sur `c1`. Compilez et testez votre méthode `main`. Que constatez-vous ?

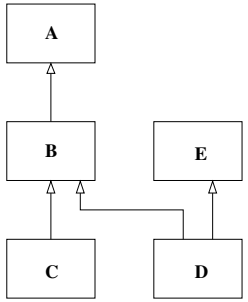
Une classe dérivée hérite d'attributs et méthodes de sa superclasse, elle peut aussi définir ses propres variables et méthodes membres, mais elle peut également *redéfinir* des méthodes héritées. Elle le fait lorsqu'elle désire modifier l'implémentation d'une méthode d'une classe parent, en particulier pour l'adapter son action à des besoins spécifiques. C'est le cas ici pour la méthode `quiSuisJe` à redéfinir dans `Carre`.

exercice 8) Dans la classe `Carre`, redéfinissez la méthode `quiSuisJe` pour qu'elle renvoie la chaîne de caractères *"un carré"*.

exercice 9) Recompilez et exécutez votre programme pour obtenir le résultat attendu.

4 Graphe d'héritage

L'héritage ne se limite pas à deux classes. Par exemple, si une classe `B` hérite d'une classe `A`, une classe `C` peut hériter de `B`, et donc par transitivité de classe `A`. Une classe `D` peut aussi hériter de `B` et d'une classe `E` créant ainsi le *graphe d'héritage* suivant :



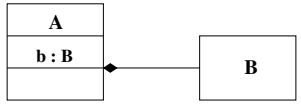
L'héritage est dit *simple*, lorsque qu'une classe dérivée ne peut posséder qu'une *seule* superclasse, et il est dit *multiple* lorsqu'elle peut en posséder *plusieurs*, comme par exemple la classe `D` de la figure précédente.

Certains langages objets ne proposent que l'héritage simple, comme par exemple le langage Java. D'autres proposent l'héritage simple et l'héritage multiple, comme C++. Nous verrons plus tard cette notion.

Notez que dans cas de l'héritage simple, le graphe d'héritage se réduit à une simple arborescence.

Autres relations

La relation d'héritage est à ne pas confondre avec d'autres relations, *agrégation* ou *composition*, qui lient les classes, comme par exemple, lorsque qu'une classe `A` possède une variable membre de type `B`. On peut représenter cette relation par le schéma suivant :



Nous voulons compléter notre application de traitement de figures géométriques.

exercice 10) Sur le modèle des classes `Rectangle` et `Carre`, écrivez la classe `Ellipse` et sa classe dérivée `Cercle`. On rappelle que pour une ellipse de petit rayon a, et de grand rayon b, le périmètre est environ égal à $\pi\sqrt{2(a^2+b^2)}$ (formule d'Euler) et la surface πab.

Les quatre classes `Rectangle`, `Carre`, `Ellipse` et `Cercle` sont des figures géométriques, et il peut être judicieux de définir une classe de base `Figure` pour les représenter.

exercice 11) Dessinez le graphe d'héritage de ces 5 classes et écrivez la classe `Figure` dans laquelle vous définirez la méthode `quiSuisJe`.

exercice 12) Dans la fonction `main` déclarez une variable `f` de type `Figure` et appliquez la méthode `quiSuisJe`.

5 Polymorphisme et liaison dynamique

Associés à celui d'héritage, les concepts de *polymorphisme* et de *liaison dynamique* donnent toute sa force à la programmation par objets.

5.1 Polymorphisme

Le polymorphisme est la faculté pour une variable de désigner à tout moment des objets de types différents. En C++, le polymorphisme est contrôlé par l'héritage. C'est-à-dire qu'une variable peut désigner des objets de sa classe de déclaration, comme des objets de toutes les classes qui en dérivent. Par exemple, une variable de type `Rectangle` peut désigner un objet de type `Rectangle`, mais également de type `Carré`. Ceci est cohérent, dans la mesure où un carré est bien un rectangle (particulier). En revanche, elle ne pourra pas désigner des objets de type `Ellipse` ou `Cercle`.

Considérons un objet `a` de type `A` et un objet `b` de type `B`. La classe `B` dérive de `A`. L'affectation licite `a=b;` provoque une conversion de l'objet `b` vers le type A. Seuls les attributs de `A` sont conservés (*ceux supplémentaires de `B` sont perdus*). En revanche, si `a` et `b` sont des pointeurs ou des références, les objets ne sont pas modifiés.

exercice 13) Placez dans la fonction `main` les deux déclarations suivantes, compilez votre classe de test, et vérifiez les affirmations précédentes.

```
Rectangle r1 = Carre(8);
Rectangle r2 = Cercle(8);
```

Compilez la classe de test. Quelle est la signification du message d'erreur ?

exercice 14) Remplacez la déclaration de la variable `r2` par :

```
Carre r2 = Rectangle(2,8);
```

exercice 15) Remplacez la déclaration de la variable `r2` par :

```
Carre r2 = r1;
```

Compilez la classe de test. Pourquoi le compilateur indique-t-il une erreur alors que `r1` désigne une objet de type `Carre` ? Faites la conversion explicite de type demandée.

5.2 Liaison dynamique

exercice 16) Dans votre fichier `main.ccp`, ajoutez la fonction suivante :

```
void afficher(const Figure *f) {
  std::cout << "Je suis " << f->quiSuisJe() << std::endl;
}
```

puis dans votre fonction `main`, testez le code suivant :

```
Rectangle *r = new Carre(8);
afficher(r);
Ellipse *c = new Cercle(8);
afficher(c);
```

Est-ce que résultat obtenu est satisfaisant ? En toute bonne logique, que devriez-vous obtenir ?

Le problème vient du fait que la méthode à appliquer est déterminée de façon *statique*, c'est-à-dire à la compilation. Ainsi, puisque dans la fonction `afficher` le paramètre est de type `Figure *`, la méthode `quiSuisJe` choisie par le compilateur est celle de la classe `Figure` et non pas celle du paramètre effectif qui n'est connu que de façon dynamique, c'est-à-dire à l'exécution du programme.

La solution à ce problème réside dans le mécanisme de *liaison dynamique* présent dans tous les langages à objets. Les langages mettent en place ce mécanisme de telle sorte que la méthode appliquée soit celle de l'objet qui est *effectivement* utilisé au moment de son exécution.

5.2.1 Méthode virtuelle

En C++, les méthodes sur lesquelles la liaison dynamique pourra s'appliquer devront être déclarées **virtual**. Pour cela, on place le mot-clé **virtual** devant la toute première définition de la méthode. Toutes les méthodes redéfinies dans les classes dérivées deviennent alors virtuelles.

Note : une méthode virtuelle peut être spécifiée *redéfinie* à l'aide du mot-clé `override`.

exercice 17) Dans la classe `Figure`, rendez la méthode `quiJeSuis` virtuelle, recompilez votre programme et testez-le pour obtenir le comportement attendu.

D'autre part, quand le mécanisme de liaison dynamique est mis en jeu, pour que le destructeur de l'objet effectivement traité soit appliqué, il est impératif qu'il soit également déclaré **virtual** dans la superclasse initiale.

---

exercice 18) Dans la fonction `main`, déclarez une variable `vf` de type vecteur (classe `vector`) de `Figure *`, et ajoutez dans ce vecteur plusieurs rectangles, carrés, ellipses et cercles.

exercice 19) À l'aide de l'énoncé *foreach*, parcourez le vecteur pour afficher sur le sortie standard la nature de chaque figure et sa surface. Quel est le problème ?

Il faut ajouter la méthode virtuelle `surface` dans la classe `Figure`. Mais quelles instructions mettre dans le corps de cette méthode, puisque on n'est pas capable de déterminer la surface du

figure quelconque ? La solution est de ne pas l'écrire. Pour cela, on définit la méthode de façon virtuelle pure, en ajoutant `=0` à la fin du prototype de la méthode.

exercice 20) Rendez `surface` (et `perimetre`) virtuelle pure et testez votre programme.

5.2.2 Classe abstraite

Une classe qui possède *au moins* une méthode virtuelle pure est appelée *classe abstraite* et ne peut être instanciée. Le corps de la méthode virtuelle pure doit être *obligatoirement* défini dans les classes dérivées. Si une des classes dérivées n'a pas défini la méthode, et en n'obtient pas une d'une superclasse, elle deviendra elle-même abstraite et ne pourra donc pas être instanciée.

exercice 21) Écrivez la classe abstraite générique `Pile` et définissez-la comme superclasse de vos classes génériques `PileTableau` et `PileChainee`. Testez votre classe `Pile`.