enseignant : Philippe REITZ
année universitaire : 2001-2002

Programmation

unité de Travaux Pratiques

TP n°5 : C++

[Corrigé]

Exercice 1 : les étudiants sportifs

Concevoir les classes permettant de représenter des étudiants sportifs, sachant que :

- tout étudiant se caractérise par son cursus (une simple chaîne de caractères)
- tout sportif se caractérise par le sport qu'il pratique
- toute personne se caractérise par son nom

Correction

Commençons par une première version ; nous définissons la classe `unePersonne`, de laquelle nous dérivons deux classes `unÉtudiant` et `unSportif` ; chacune de ces trois classes porte un attribut qui lui est propre, de type **char***, implantant la propriété indiquée dans l'énoncé. Le code C++ correspondant ressemble à :

```
class unePersonne
{
protected:
    char* nom;
public:
    unePersonne(char* n="une personne")
        : nom(n) {};
public:
    char* sonNom() const { return nom; };
};

class unEtudiant
: public unePersonne
{
protected:
    char* cursus;
public:
    unEtudiant(char* n="un étudiant", char* c="un cursus")
        : unePersonne(n), cursus(c) {};
public:
    char* sonCursus() const { return cursus; };
};

class unSportif
: public unePersonne
{
protected:
    char* sport;
public:
    unSportif(char* n="un sportif", char* s="un sport")
        : unePersonne(n), sport(s) {};
public:
    char* sonSport() const { return sport; };
};
```

Puisque le lien d'héritage est exploité pour représenter la relation de spécialisation (un étudiant est une forme particulière de personne), il est naturel de définir la classe des étudiants sportifs comme une spécialisation des classes des étudiants et des sportifs. La classe `unÉtudiantSportif` hérite donc à la fois de `unÉtudiant` et de `unSportif` ; l'héritage est dit *multiple*. Le code semble ne pas poser de difficulté particulière, et nous serions tenté d'écrire :

```
class unEtudiantSportif
: public unEtudiant, public unSportif
{
public:
    unEtudiantSportif(char* n="un étudiant sportif",
                      char* c="un cursus",
                      char* s="un sport")
```

```
        : unEtudiant(n, c), unSportif(n, s) {};
};
```

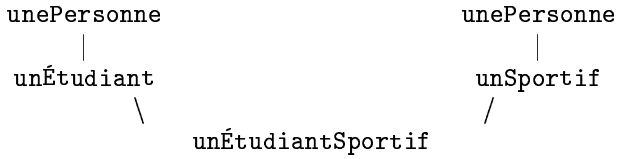
En fait, c'est moins simple qu'il n'y paraît...

Tout d'abord, notons que, dans la partie initialiation du constructeur, nous avons naturellement initialisé `unÉtudiantSportif` au travers des constructeurs de ses classes de base, `unÉtudiant` et `unSportif`, comme nous le faisions jusqu'ici dans les cas d'héritage simple.

Toutefois, un problème se pose : tout étudiant sportif initialisé via le constructeur `unEtudiantSportif(n, c, s)` est initialisé comme étudiant via `unEtudiant(n, c)`, et comme un sportif via `unSportif(n, s)`.

A priori, rien ne nous empêchait d'écrire pour cette initialisation qu'un étudiant sportif serait initialisé via `unEtudiant(c, c)` et `unSportif(s, s)`. Bien qu'absurde dans le cadre de cet exercice, cette possibilité soulève un réel problème de fond : dans ce cas, lorsque l'initialisation d'un étudiant sportif est terminée, son nom est-il `c` ou `s` ? Puisque les deux constructeurs sont nécessairement appelés l'un après l'autre, seul le dernier exécuté aura le dernier mot, i.e. donnera son nom à notre objet. Dans les spécifications de `C++`, rien n'est dit sur l'ordre d'exécution des constructeurs figurant dans une initialisation : un premier compilateur peut donc exécuter les constructeurs dans l'ordre où ils figurent dans le code source, mais un autre compilateur peut faire un choix différent.

Les concepteurs de `C++` ont résolu ce problème d'une façon très simpliste : à chaque fois qu'une classe est héritée de façon multiple (`unePersonne` dans notre exercice), elle est dupliquée de telle manière que le graphe d'héritage ne contienne plus de cycle ; dans notre cas, la hiérarchie obtenue ressemble à :



Du point de vue de la classe `unÉtudiantSportif`, cette hiérarchie montre que les propriétés de `unePersonne` sont héritées deux fois, donc qu'elles existent en double ! Par exemple, un objet étudiant sportif porte deux noms : celui introduit par la classe `unÉtudiant`, et celui introduit par `unSportif`.

Le problème d'initialisation indiqué précédemment disparait donc :

```
unEtudiantSportif::unEtudiantSportif(
                      char* n="un étudiant sportif",
                      char* c="un cursus",
                      char* s="un sport")
        : unEtudiant(c, c), unSportif(s, s) {};
```

- en tant qu'étudiant, le nom de l'étudiant sportif sera `c`
- en tant que sportif, le nom de l'étudiant sportif sera `s`

Toutefois, un nouveau problème apparaît : si `es` est un objet de la classe `unÉtudiantSportif`, alors `es.sonNom()` est ambigu : soit l'appel correspond à `es.unEtudiant::sonNom()`, soit à `es.unSportif::sonNom()` ; le compilateur étant incapable de lever automatiquement l'ambiguïté, il sera nécessaire de la lever explicitement, soit un introduisant l'opérateur de résolution de portée `::`, comme dans la phrase précédente, soit en redéfinissant la méthode `sonNom` dans la classe `unÉtudiantSportif`.

Le code final ressemble alors à :

```
class unEtudiantSportif
: public unEtudiant, public unSportif
{
public:
    unEtudiantSportif(char* n="un étudiant sportif",
                      char* c="un cursus",
                      char* s="un sport")
        : unEtudiant(n, c), unSportif(n, s) {};
public:
    char* sonNom() { return unEtudiant::sonNom(); };
};
```

Cette solution, bien que désormais fonctionnelle, n'est pas véritablement satisfaisante, puisque nous souhaiterions que tout étudiant sportif n'ait bien qu'un seul nom.

Les concepteurs de C++ ont permis qu'un tel choix puisse s'exprimer : la classe unePersonne sera déclarée *classe virtuelle* [1].

Dès lors, la duplication de la classe unePersonne n'est plus réalisée : les propriétés héritées plusieurs fois seront bien les mêmes, i.e ne seront pas dupliquées.

Pour signaler que la classe unePersonne est virtuelle, nous devons récrire la définition des classes unÉtudiant et unSportif ainsi :

```
class unEtudiant
: virtual public unePersonne
{
  ... // code inchangé
};

class unSportif
: virtual public unePersonne
{
  ... // code inchangé
};
```

Noter l'aberration de la solution : la spécification de la virtualité de la classe unePersonne n'est pas attachée à la classe elle-même, mais aux classes qui en héritent ! Toute classe déclarée virtuelle pour une classe l'est aussi pour toutes ses sous-classes.

Cela ne résout toujours pas notre problème : en effet, le constructeur de la classe unÉtudiantSportif est, pour l'instant, défini ainsi :

```
unEtudiantSportif::unEtudiantSportif(
                      char* n="un étudiant sportif",
                      char* c="un cursus",
                      char* s="un sport")
        : unEtudiant(n, c), unSportif(n, s) {};
```

Le problème signalé précédemment est toujours présent : si l'initialisation était définie ainsi :

```
unEtudiantSportif::unEtudiantSportif(
                      char* n="un étudiant sportif",
                      char* c="un cursus",
                      char* s="un sport")
        : unEtudiant(c, c), unSportif(s, s) {};
```

nous serions toujours dépendant de l'ordre d'exécution des constructeurs. L'attribut nom n'étant présent qu'une fois, le problème semble donc insoluble. Là encore, les concepteurs de C++ l'ont résolu d'une manière abominable : puisqu'il y a un problème d'initialisation sur les classes virtuelles, alors autant écrire explicitement comment les constructeurs de ces classes doivent être appelés. Pour notre exercice, cette solution se traduit par une redéfinition du constructeur de unÉtudiantSportif :

```
unEtudiantSportif::unEtudiantSportif(
                      char* n="un étudiant sportif",
                      char* c="un cursus",
                      char* s="un sport")
        : unEtudiant(n, c), unSportif(n, s), unePersonne(n) {};
```

Autrement dit, la partie initialisation d'un constructeur doit mentionner :

- l'initialisation des membres données (attributs) introduits par la classe
- l'appel aux constructeurs des classes de base (classes directement héritées, figurant dans la définition de la classe)
- l'appel aux constructeurs des classes virtuelles héritées.

1. Attention, une classe virtuelle n'a absolument rien à voir avec une fonction membre virtuelle. Il semble que le choix de ce terme soit lié à des considérations d'économie : afin de ne pas introduire trop de nouveaux mots-clé dans C++, ses concepteurs ont jugé que virtual, exploité pour les fonctions membres, pourrait aussi l'être pour les classes, les contextes d'usage étant suffisemment distincts.

La fameuse modularité de la programmation objet est donc singulièrement remise en cause en C++, dès lors qu'il existe des classes virtuelles !

L'appel au constructeur de unePersonne pour unÉtudiantSportif masque les appels à ce même constructeur dans les classes unÉtudiant et unSportif. Pour démontrer ce principe de masquage, supposons que nous ayons défini les constructeurs comme ceci (absurde, mais démonstratif) :

```
unePersonne::unePersonne(char* n="une personne")
        : nom(n) {};

unEtudiant::unEtudiant(char* n="un étudiant", char* c="un cursus")
        : unePersonne(c), cursus(c) {};

unSportif::unSportif(char* n="un sportif", char* s="un sport")
        : unePersonne(s), sport(s) {};

unEtudiantSportif::unEtudiantSportif(char* n="un étudiant sportif",
                char* c="un cursus", char* s="un sport")
        : unEtudiant(s, c), unSportif(c, s), unePersonne(n) {};
```

alors tout étudiant se voit attribuer c pour nom, tout sportif se voyant attribuer s, et tout étudiant sportif n ! Lors de l'initialisation d'un étudiant sportif, il est bien initialisé tout d'abord en tant qu'étudiant (unEtudiant(s, c)), qui implique qu'il est initialisé comme une personne, mais pas via l'appel au constructeur unePersonne(c) comme le préconise la classe unÉtudiant, mais unePersonne(n), comme le précise la classe unÉtudiantSportif.

Pour notre exercice, nous n'avons plus le problème d'ambiguïté sur la méthode sonNom(), comme dans la première version, puisque unePersonne n'est plus dupliquée.

La version finale de notre classe est donc, pour cette solution avec classe virtuelle :

```
class unEtudiantSportif
: public unEtudiant, public unSportif
{
public:
    unEtudiantSportif(char* n="un étudiant sportif",
                      char* c="un cursus",
                      char* s="un sport")
        : unEtudiant(n, c), unSportif(n, s), unePersonne(n) {};
};
```

Exercice 2 : les figures (suite)

En reprenant les développements issus de l'exercice sur les figures du TP précédant, écrire une classe permettant de représenter des figures pesantes, lesquelles sont caractérisées par une densité.

Connaissant la densité et la taille d'une figure pesante, il est possible d'en déduire son poids.

Définir alors la classe des cercles pesants.

Correction

Cette classe ne pose guère de difficultés :

```
class uneFigurePesante
: public uneFigure {
protected:
  float densite;
public:
  uneFigurePesante(float, float, float);
protected:
  char* intitule_genre() const {
    return "une figure pesante"; };
  virtual char* intitule_densite() const {
    return ""; };
public:
```

```
  void se_decrire() const;
  virtual float poids() const { return densite*taille();}
};

uneFigurePesante::uneFigurePesante(float X, float Y, float d)
  : uneFigure(X, Y), densite(d) {};

void uneFigurePesante::se_decrire() const {
     uneFigure::se_decrire();
     cout << "Ma densité " << intitule_densite() << " est "
                          << densite << endl
          << "Mon poids est " << poids() << endl; };
```

Notre définition de la méthode de description anticipe sur la suite : en effet, si une figure pesante est une figure plane, alors sa densité est de type surfacique ; s'il s'agit d'une figure linéique, la densité est dite linéique ; pour une figure volumique, la densité est dite volumique. D'où l'introduction d'une méthode `intitule_densite`.

Un cercle pesant est un cas particulier de cercle, lequel est un cas particulier de figure plane. Nous choisissons de définir la classe des figures planes et pesantes, de laquelle nous dévirerons celle des cercles pesants.

Une figure plane et pesante hérite donc des figures planes et des figures pesantes ; nous sommes dans un cas d'héritage multiple. Cette classe est abstraite, puisque nous sommes toujours incapables de définir la notion de `taille` à ce niveau.

L'héritage multiple standard implique que la classe `uneFigure` sera dupliquée autant de fois qu'elle sera héritée par une sous-classe ; ces sous-classes risquent donc de se voir associer plusieurs points de référence (propriété de `uneFigure`), alors qu'il ne faudrait qu'un seul point de référence par figure plane et pesante. Ceci nous oblige à faire de `uneFigure` une classe virtuelle, donc à récrire une partie des définitions déjà étudiées :

```
class uneFigurePlane
: virtual public uneFigure {
  ... // contenu inchangé
};

class uneFigurePesante
: virtual public uneFigure {
  ... // contenu inchangé
};
```

Notre définition des figures planes et pesantes commencera donc ainsi :

```
class uneFigurePlanePesante
: public uneFigurePlane, public uneFigurePesante
{
  ...
```

Un examen des conflits d'héritage montre qu'il y a conflit :

– de masquage pour : `taille`, `intitule_genre` et `intitule_taille`.

– de multiplicité pour : `se_decrire`.

Ces quatre méthodes devront être redéfinies, au moins pour lever ces conflits. Nous aboutissons à la définition suivante :

```
class uneFigurePlanePesante
  : public uneFigurePlane, public uneFigurePesante
{
public:
  uneFigurePlanePesante(float X, float Y, float d);
protected:
  char* intitule_genre() const {
    return "une figure plane et pesante"; };
  char* intitule_densite() const
    return "surfacique"; ;
public:
  void se_decrire() const;
```

```
// redéfinition de méthodes pour résoudre tous les conflits de multiplicité
protected:
  char* intitule_taille() const {
    return uneFigurePlane::intitule_taille(); };
public:
  float taille() const {
    return uneFigurePlane::taille(); };
};

uneFigurePlanePesante::uneFigurePlanePesante(float X, float Y, float d)
    : uneFigurePlane(X, Y), uneFigurePesante(X, Y, d), uneFigure(X, Y)
{};

void uneFigurePlanePesante::se_decrire() const {
    uneFigurePlane::se_decrire();
    uneFigurePesante::se_decrire();
};
```

Dès lors, tout est en place pour définir ce qu'est un cercle pesant : il s'agit d'un cercle qui est aussi une figure plane et pesante. L'examen des conflits d'héritage montre qu'il y a des conflits :

- de masquage pour : `surface`, `taille`, `intitule_taille`.
- de multiplicité pour : `se_decrire`.

Leur résolution nous amène à la définition suivante :

```
class unCerclePesant
: public unCercle, public uneFigurePlanePesante
{
public:
  unCerclePesant(float X, float Y, float R, float D);
public:
  char* intitule_genre() const { return "un cercle pesant"; };
public:
  // redéfinition de méthodes pour résoudre tous les conflits d'héritage
  float surface() const { return unCercle::surface(); };
  void se_decrire() const { uneFigurePlanePesante::se_decrire(); };
};

unCerclePesant::unCerclePesant(float X, float Y, float R, float D)
    : unCercle(X, Y, R), uneFigurePlanePesante(X, Y, D), uneFigure(X, Y)
{};
```

Noter la définition plus que contestable, dans un paradigme objet, du constructeur de `unCerclePesant` : il y a un appel aux 2 constructeurs des classes de bases directes, puis un appel à celui de la classe virtuelle `uneFigure`.