Polymorphisme en C++

M2103 Programmation Objets en C++

Dept Info - IUT Bordeaux - 2016

Table des matières

Révision 1 du 24 mars 2016

1 Le(s) polymorphisme(s)

Définition : le polymorphisme en programmation, c'est l'idée de disposer d'une interface unique pour manipuler des objets de type différents

Il y a 3 formes : le polymorphisme ad hoc, par généricité, et par sous-typage (dit aussi polymorphisme d'inclusion)

Quelques mots sur chacun :

— le **polymorphisme ad hoc**, c'est quand on peut utiliser le même nom de fonction pour des paramètres différents. C'est la surcharge (*overloading*) permise en C++, où on peut définir deux fonctions ayant le même nom

```
void f(string s) {
...
}

void f(int n) {
...
}
```

et on pourra l'appeler :

```
f("coucou");
f(12);
```

c'est le compilateur qui déterminera, au vu des types des paramètres, si il faut appeler `void f(string)` ou `void f(int)`.

— **le polymorphisme paramétré** : vous savez sans doute qu'il existe une fonction `max` que vous pouvez utiliser ainsi

```
string a, b;
...
cout << max(a, b);
cout << max(12,34);
```

en fait, `max` est une **fonction générique**, c'est-à-dire une famille de fonctions. En l'occurrence, vous utilisez les fonctions `max<string>` et `max<int>`, mais comme le compilateur C++ est malin, il déduit le type dont il s'agit et vous évite d'écrire explicitement :

```
cout << max<string> (a, b);
cout << max<  int > (12,34);
```

Quoi qu'il en soit, `max` n'est pas une fonction mais un "template" pour une *famille* de fonctions `max<T>`, pour tout type `T` possédant l'opérateur de comparaison "<".

— enfin le **polymorphisme par sous-typage** caractéristique de la programmation orientée objets : l'interface est définie pour une classe d'objets et est héritée par les sous-classes qui en sont dérivées.

Par exemple, la classe *Shape* de la SFML déclare des opérations `setPosition()`, `setFillColor()`, `move()` qui pourront être utilisées sur toutes ses sous-classes : `CircleShape`, `RectangleShape`, `ConvexShape` ...

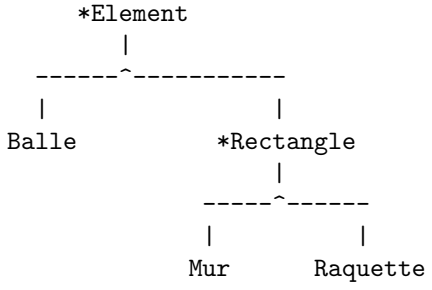
2 Objectif, hiérarchie de classes

L'objectif de la programmation orientée objets, c'est de limiter l'écriture de code qui fait *presque* la même chose.

Considérons par exemple un jeu comme Pong, où il y a des raquettes, des balles et des murs. Dans la programmation, il va falloir détecter la collision d'une balle et une raquette (rebond), un mur du fond (fin), et un mur latéral.

Dans la mesure où les murs et les raquettes peuvent être considérés comme des rectangles, on aura tout intérêt à appeler une même fonction qui détecte la collision entre une balle et un rectangle, plutot que d'en faire deux versions.

Donc nous voyons les murs et les raquettes comme deux types particuliers d'éléments rectangulaires, et nos types de données forment une hiérarchie de classes :



En pratique, dans le jeu, on n'a que des instances de `Balle`, `Mur` et `Raquette` : ce sont des **classes concrètes**. Les autres classes `Element` et `ElementRectangulaire` n'ont pas d'instances : ce sont des **classes abstraites** qui déclarent les champs et les actions applicables aux instances de leurs sous-classes respectives.

— Toute balle, raquette, tout mur a des coordonnées et peut être dessiné : il y a des membres x, y et une fonction commune `dessiner()` applicable à tout élément, donc définies dans la classe `Element`, qui est la **classe de base** de la hiérarchie.

— Les murs et les raquettes ont une largeur et une hauteur : `w` et `h` sont définis dans `Rectangle`, qui dérive (ou hérite) de `Element`.

Par contre, si la fonction `dessiner()` est déclarée dans l'interface des `Eléments`, son comportement diffère selon les sous-classes :

— pour une `Balle` on trace un cercle,
— pour un `Mur` ou une `Raquette` un rectangle.

La fonction `dessiner()` sera donc redéfinie (**overriding**) dans la classe concrète `Balle`, et dans la classe abstraite `Rectangle` qui fera le travail pour ses sous-classes concrètes.

Ceci dit, dans `Element` la fonction `dessiner()` sera définie, mais pas *implémentée*

3 Code de l'exemple

3.1 Éléments

Ci-dessous la déclaration de la classe `Element`, qui est à la base de la hiérarchie. Pour l'instant, pour ne pas compliquer les explications, tout est public.

```
class Element
{
public:
   int _x, _y;
   Element(int x, int y);
   virtual void dessiner() const = 0;
   virtual ~Element()      = default;
};
```

Commentaires :

— les raquettes, les murs et les balles auront tous un `x` et un `y` : on les définit dans `Element` qui est la classe la plus générale ;
— de même il faudra savoir les dessiner ;
— le mot clé `virtual` indique qu'une fonction **peut** être rédéfinie dans les sous-classes ;
— nous verrons pourquoi le destructeur de la classe de base **doit** être virtuel ;
— la spécification `= 0` dit que la fonction `dessiner()` **doit** être redéfinie pour les sous-classes instanciables. Dans le jargon C++, on dit que c'est une fonction **virtuelle pure**.

En effet, `dessiner()` doit être applicable à toute instance, mais au niveau de la classe `Element`, nous ne savons évidemment pas quoi faire pour les tracer.

Ici nous n'avons donc que le constructeur à écrire

```
Element::Element(int x, int y)
   : _x {x}, _y{y}
 {}
```

3.2 Balles

La **définition de la classe** dit qu'une `Balle` est un `Element` qui a un rayon, et qu'on peut dessiner.

```
class Balle : public Element
{
public:
     int _r;
     Balle(int x, int y, int r);
     virtual void dessiner() const override;
};
```

- Le mot-clé `override` demande au compilateur de vérifier qu'on redéfinit une fonction qui existe bien "au dessus". C'est une précaution (facultative, mais très utile) introduite par C++11.
- le mot-clé `virtual` est facultatif ici, puisque la redéfinition d'une fonction virtuelle est virtuelle.

La fonction `dessiner()` étant redéfinie dans `Balle`, cette classe n'a plus de fonctions pures : c'est une classe concrète qui peut avoir des instances.

Les **définitions de fonctions-membres :**

```
Balle::Balle(int x, int y, int r)
  : Element {x,y }
  , _r {r}
 {}

void Balle::dessiner() const {
   // code pour dessiner un cercle
   // de centre _x, _y et rayon _r
}
```

- Comme une `Balle` **est un** `Element`, la liste d'initialisation du constructeur fait appel au constructeur de la classe `Element`
- pour dessiner la balle, on utilise les coordonnées qui sont ici des données membres publiques de `Element`. Si elles étaient privées, on ne le pourrait pas.

Il existe un moyen terme entre `public` (membre visible par tout le monde) et `prive` (visible uniquement dans le code de la classe), qui s'appelle `protected` et signifie : visible dans la classe et dans les sous-classes.

3.3 Rectangles

La classe `Rectangle` est similaire : elle est également dérivée d'`Element`

```
class Rectangle : public Element
{
public:
    int _w, _h;
    Rectangle (int x, int y,
               int w, int h);
    void dessiner() const override;
};
```

```
Rectangle::Rectangle (int x, int y,
                      int w, int h)
  : Element {x,y }
  , _w {w}
  , _h {h}
{}

void Rectangle::dessiner() const
{
    // code pour dessiner un rectangle
}
```

Au niveau de la conception de la hiérarchie, elle est abstraite (on n'est pas censé en créer des instances), mais du point de vue du langage C++, elle ne l'est pas, parce qu'elle n'a pas de fonctions virtuelles pures.

Un programmeur distrait pourrait donc créer des instances de `Rectangle` sans que le compilateur le signale : nous verrons plus loin comment rendre cette classe abstraite.

3.4 Murs et Raquettes

Enfin, les murs et les raquettes héritent de `Rectangle`, les raquettes ont de plus des fonctions spécifiques

```
class Mur : public Rectangle
{
public:
    Mur (int x, int y,
         int w, int h);
};

Mur::Mur(int x, int y,
         int w, int h)
   : Rectangle {x, y, w,h }
{}

class Raquette : public Rectangle
{
public:
    Raquette (int x, int y,
              int w, int h);
    void monter();
    void descendre();
};

Raquette::Raquette(int x, int y,
                   int w, int h)
   : Rectangle {x, y, w,h }
{}


void Raquette::monter() {
   // code pour monter
}

void Raquette::descendre() {
   // code pour descendre
}
```

3.5 Code client

et voici un exemple de code qui utilise ces classes

```
int main()
{
    Balle    balle    {300, 300,  10};
    Raquette raquette {100, 200,  20,  50};
    Mur      gauche   {  0,   0,  50, 600};
    Mur      droite   {750,   0,  50, 600};
    Mur      haut     {  0,   0, 800,  50};
    Mur      bas      {  0, 550, 800,  50};

    vector<Element *> elements {
         & balle,  & raquette,
         & gauche, & droite,
         & haut, & bas
    };

    raquette.monter();

    for (Element *p: elements) {
       p->dessiner();
    }

    return EXIT_FAILURE;
}
```

Remarquez qu'on ne peut pas définir un `vector<Element>`, parce qu'`Element` est une classe abstraite, qui n'a pas d'instances.

Par contre on peut avoir des pointeurs de type `Element*`, et ces pointeurs peuvent contenir l'adresse d'instances de n'importe quel sous-type d'`Element`.

Ici vous voyez donc le **polymorphisme par sous-typage** : dans la boucle le pointeur `p` désigne des éléments de divers types dérivés d'`Element`, et ce pointeur permet d'appeler la fonction `afficher()` qui fait partie de l'interface d'`Element`.

4 Fonctions virtuelles : résolution dynamique

La question qui se pose, dans l'exemple précédent, est de savoir quelle fonction est appelée quand s'exécute l'instruction

```
p->dessiner();
```

La réponse est que ça dépend.

Au vu du type *déclaré* pour `p` (`Element*`), on pourrait penser que c'est la fonction `Element::dessiner()`. Mais en fait non. Comme elle a été déclarée `virtual`, la fonction appelée va dépendre du type effectif de l'objet pointé par `p`. C'est-à-dire que

- si `p` contient l'adresse d'une `Balle`, ça appelle `Balle::dessiner()`
- si `p` contient l'adresse d'une `Raquette`, ça appelle `Rectangle::dessiner()`, parce que `Raquette` ne redéfinit pas `dessiner()`, et donc la fonction `dessiner()` pour les balles est héritée de `Rectangle`.

Pour que ça marche, le compilateur code une indication de type dans les objets qui ont des fonctions virtuelles.

Autrement dit, ce n'est qu'au moment de l'exécution qu'on peut savoir quel code va être exécuté. C'est ce qu'on appelle la **résolution dynamique** (en regardant le type codé dans l'objet) par opposition à la **résolution statique** (en regardant le type déclaré pour le pointeur ou la variable)

5 Conséquences du polymorphisme

Considérons la hiérarchie de classes :

```
class A {
public:
    X _x;
    virtual void f() = 0;
    virtual ~A();
};

class B {
public:
    Y _y;
    virtual void f() override;
    void g();
    virtual ~B();
};

class C {
public:
    Z _z;
    virtual void f() override;
    void h();
    virtual ~C();
};
```

5.1 Pourquoi le destructeur doit être virtuel

Examinons ce code qui effectue des allocations dynamiques, et bien sûr, des libérations :

```
vector<A*> v;
v.push_back(new B{});
v.push_back(new C{});
...
for (A* p : v) {
   delete p;
}
```

Lors du `delete`, il faut appeler le bon destructeur : un coup celui de `B`, un coup celui de `C`. Et pour cela on ne peut que se baser sur le type de l'objet : il doit y avoir résolution dynamique, ce qui ne se produit que si le destructeur est virtuel.

Si on oublie de déclarer le destructeur comme virtuel, il y a résolution statique et c'est le destructeur de `A` qui est toujours appelé (parce que `p` est de type `A*`). Autant dire que ça ne va pas le faire.

Donc **règle générale** : toujours déclarer `virtual` le destructeur de la classe de base d'une hiérarchie polymorphe

5.2 Destructeur virtuel pur

Contrairement à ce qu'on peut lire parfois, une fonction pure *n'est pas* une fonction qui n'est pas implémentée.

Du point de vue de C++, une fonction pure c'est une fonction qui est déclarée avec la spécification `= 0`. Et une fonction pure **peut ne pas avoir de définition**. Elle peut aussi en avoir.

La spécification `= 0` interdit simplement qu'on l'appelle *explicitement* (a fortiori, si elle n'est définie, on ne voit pas comment on ferait !)

Ceci nous est utile pour la classe `Rectangle` qu'on voudrait abstraite, mais qui n'a aucune fonction virtuelle pure, puisqu'elle redéfinit `dessiner()`.

Le "truc" pour rendre `Rectangle` abstraite est de redéfinir son destructeur en le rendant virtuel pur (en fait il est déja virtuel parce que celui de `Element` l'est).

```
class Rectangle {
   ...
   virtual ~Rectangle() = 0;
   ...
};
```

Et commme nous savons les destructeurs des sous-classes `Mur` et `Raquette` appellent *implicitement* celui de `Rectangle`, nous devons fournir son code

```
Rectangle::~Rectangle()
{}
```

Résumé : quand on définit des hiérarchies qui utilisent le polymorphisme :

- déclarer obligatoirement virtuel le destructeur de la classe de base
- rendre pur le destructeur de toutes les classes abstraites.

5.3 Affectation et *slicing*

Dans du code polymorphe, où la même variable peut désigner des données de types différents, les objets sont nécessairement manipulés par l'intermédiaire de pointeurs ou de références.

En effet, on ne connait pas leur taille, et une variable du type le plus général (`A`) risque d'être trop petite pour accueillir une donnée d'un type dérivé.

Dans l'exemple ci-dessus, un `B` est un `A` à qui on a rajouté un champ `_y` de type `Y`. Autrement dit il va être plus gros qu'un `A`.

Attention, en C++ il est tout à fait légal d'écrire une affectation comme celle-ci :

```
A a;
B b;

a = b;
```

mais il faut bien comprendre ce qui se passe exactement : le champ `_x` de `b` est copié dans celui de `a`, mais pas le champ `_y` qui n'existe pas dans la classe `A`.

C'est ce qu'on appelle le **slicing** : ce qui est affecté, c'est une partie tronçonnée de `b` pour que ça rentre dans `a`.

En général, ce n'est pas ce qu'on veut faire.

A priori, **l'affectation en sens inverse** (`b = a`) n'a pas de sens, parce que le compilateur ne peut pas inventer les valeurs des champs manquants. Si on voulait lui donner un sens, il faudrait définir l'affectation dans la classe `B` :

```
class B {
   ...
   B &  operator=(const A &);
   ...
}
```

5.4 Manipulation via des pointeurs et des références

Donc **règle générale** : quand on manipule des données polymorphes, on le fait à travers d'**adresses**, soit explicitement (pointeurs), soit implicitement (références).

Exemple d'utilisation de références

```
void foo(A & a)  // par référence
{
   a.f();       // appel fonction virtuelle
}
...

B b;
```

```
C c;
foo(b);
foo(c);
```

Les deux appels de `foo` n'amènent pas de *slicing* (il n'y pas copie lors du pasage par référence), et l'appel de la fonction virtuelle exécute donc, selon les cas, soit `B::f()`, soit `C::f()`.

6 Transtypage dynamique

Nous avons vu plus haut des exemples où on affectait dans un pointeur une adresse d'une donnée d'un sous-type ? Exemple :

```
A * p = new B{};
```

et c'est la base du polymorphisme en C++.

Dans certains cas on souhaite faire la conversion inverse, par exemple :

```
A FAIRE :
    Soit p un pointeur vers A
    Tester si p pointe sur un B
    et si oui, lancer la fonction g()
```

Or la fonction `g()` n'est pas dans l'interface de la classe `A`. Il s'agit donc, en réalité, de contourner un problème de conception puisque

- on souhaite utiliser le polymorphisme, c'est à dire traiter les objets avec une interface commune (même fonctions)
- et en même temps on n'a pas prévu les fonctions qui conviennent pour ce traitement commun applicable à tous.

Le remède serait, par exemple, d'avoir défini dans `A` une fonction `g()` qui, par défaut, ne fait rien.

Bref, il se trouve que, indépendamment du bien-fondé de son utilisation, on dispose en C++ d'une fonction générique `dynamic_cast` de conversion de pointeurs. On l'utilise comme ceci :

```
A *pa = ......

B *pb = dynamic_cast<B*> (pa); // 1 conversion
if (pb) {                      // 2 réussi ?
   pb -> g();                  // 3 appel g()
}
```

Explications :

1. la fonction `dynamic_cast<B*>`
   - prend comme paramètre un pointeur `pa`
   - examine le type de l'élément pointé
   - si cet élément est de type `B` (ou un sous-type), retourne son adresse dans un pointeur de type `B*`
   - sinon retourne le pointeur nul.
2. le test vérifie que le pointeur n'est pas nul
3. et la fonction `g()` est exécutée.

C'est ce qu'on appelle un "downcasting", parce qu'on va vers un type qui est plus bas dans la hiérarchie de classes : à partir d'un `A*`, on obtient un `B*`.

7 Visibilité "protected"

Intérmédiaire entre `private` et `public`, l'attribut `protected` indique qu'un membre (donnée ou classe) ne peut être appelé que depuis la classe elle-même ou une de ses sous-classes.

Le choix entre `protected` et `private` pour les membres non privés est une question de définition de l'interface que nous voulons présenter pour l'écriture de sous-classes, qui est différente de l'interface "publique" pour le code client.

Par exemple, si on écrit

```
class Element
{
protected:
   int _x, _y;

public:
   Element(int x, int y);
   void setXY(int x, int y);
   void getXY(int &x, int &y);
   virtual void dessiner() const = 0;
   virtual ~Element()      = default;
};
```

- les programmeurs de code client (utilisateurs de classes) devront obligatoirement passer par les accesseurs `setXY()` et `getXY()` pour utiliser les coordonnées d'un élément,
- mais les programmeurs de sous-classes pourront travailler directement avec les `_x` et `_y` d'un élément.

8 Complément : relations entre classe et sous-classe.

Reprenons notre exemple avec une classe `A` qui a un membre `x`, et `B` qui y ajoute un `_y`.

Nous allons voir comment écrire les constructeurs, l'affectation etc. de la sous-classe `B` en nous appuyant sur ceux de `A`.

8.1 Le constructeur paramétré

Si `A` possède un constructeur `A::A(const X &x)`, que faire dans le constructeur pour `B` qui a un paramètre supplémentaire ?

```
class B {
  ...
  B (const X & x, const Y & y);
  ...
};
```

Nous avons déjà vu ce cas : la plupart du temps, on suffira d'invoquer un constructeur de `A` dans la liste d'initialisation des membres :

```
B::B (const X & x, const Y & y)
   :   A(x)                  // constructeur de A
   ,  _y{y}
{
...
}
```

8.2 Le constructeur par copie

Même solution pour le constructeur par copie

```
B::B (const B & b)
   :   A(b)         // constructeur par copie de A
   ,  _y{b.y}       // le membre _y de b
{
...
}
```

Remarquez qu'on passe un `B` comme paramètre par référence au constructeur de `A`, qui est donc polymorphe sans être virtuel : il peut copier tout type dérivé de `A`.

8.3 L'affectation

Dans `B` l'affectation peut s'écrire

```
B &  B::operator= (const B & b)
{
   A::operator=(b);
   _y = b.y;
}
```

Rappelez-vous d'abord que `x = y` peut s'écrire `operator=(x,y)` ou `x.operator=(y)` selon que `operator=` a été définie comme fonction libre ou comme fonction membre de la classe de `x`.

Le préfixe `A::` dans `A::operator=` indique que l'on appelle l'affectation définie pour la classe `A`, et non celle de `B`.

Elle est appliquée à l'objet courant, on pourrait aussi écrire explicitement `(*this).A::operator=(b);`.

Bref, on appelle l'affectation de `A`, qui va copier les champs de la partie `A` de l'objet `b` (passé par référence, comme précédemment) qui est de classe `B`.

8.4 Le destructeur

Il n'y a pas à invoquer explicitement le destructeur de `A` puisque c'est fait automatiquement après avoir exécuté le corps qui s'occupe de libérer les ressources que détient l'instance de `B`

```
B::~B() {
   // corps
   // libérer ce dont s'occupe le B
}
```

8.5 Invoquer une fonction redéfinie

Dans `B`, si une fonction de `A` a été redéfinie, il faut ajouter un préfixe pour préciser si on veut appeler la fonction de `A` et pas la redéfinition de `B`. Exemple d'utilisation

```
class A {
....
public:
   virtual void afficher() const;
}

class B {
....
public:
   virtual void afficher() const override;
}
```

Dans `A::afficher()` on fait afficher `_x`

```
void A::afficher() const {
   cout << "x=" << _x;
}
```

et dans `B::afficher()` on fait aussi apparaître `_y`

```
void B::afficher() const {
   A::afficher();                // appel
   cout << ", y=" << _y;
}
```

8.6 Exercice

Étudiez cette variante sur le même thème :

```
class A {
....
protected:
    virtual string nom_type()   const;
    virtual string parametres() const;
public:
   void afficher() const;  // non virtuelle
}

class B {
....
protected:
    virtual string nom_type()   const override;
    virtual string parametres() const override;
}


void A::afficher() const {
   cout << nom_type()
        << "(" << parametres() << ")";
}

void A::nom_type() const {
    return "A";
}

string A::parametres() const {
    return "x=" + to_string(_x);
}

void B::nom_type() const {
    return "B";
}

string B::parametres() const {
    return A::parametres()
           + ", y=" + to_string(_y);
}
```

La fonction publique `afficher()` est non virtuelle, et elle utilise deux fonctions auxiliaires protégées qui sont définies par polymorphisme.