Polymorphisme d'inclusion [pm]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

1 Polymorphisme **3**
1.1 Définitions . . . 3
1.2 Liaisons statiques et dynamiques . . . 4

2 Méthode virtuelle **5**
2.1 Déclaration d'une méthode virtuelle . . . 5
2.2 Méthode virtuelle – Attention Piège . . . 6
2.3 Méthode virtuelle – Technique . . . 7

3 Collections hétérogènes **8**
3.1 Collection hétérogène . . . 8
3.2 Pointeurs et intégrité des données . . . 10

4 Compléments **11**
4.1 Forme canonique de Coplien . . . 11
4.2 Les tables de méthodes virtuelles . . . 12

5 C++ : Spécificités **14**

6 Polymorphisme ([Drouillon-CC1 :c7], Juillet 2017) **14**
6.1 Qu'est ce que le polymorphisme . . . 14
6.2 Accès pointeur par défaut aux méthodes redéfinies . . . 14
6.3 Accès pointeur aux méthodes virtuelles redéfinies . . . 15

C++ - Polymorphisme d'inclusion

Mots-Clés Polymorphisme d'inclusion ■
Requis Classes, Classes (suite), Pointeurs, Héritage ■
Difficulté ●●○ (30 min) ■

Introduction

Ce module traite du **polymorphisme d'inclusion**.

1 Polymorphisme

1.1 Définitions

Polymorphisme [1]
Capacité pour une entité de prendre plusieurs formes.

Exemple
Certains animaux en font leur spécialité comme les abeilles : chez une même espèce, il y a trois formes d'individus, la femelle (reine), les mâles (faux-bourdon) et les ouvrières (femelles stériles). Trois formes différentes pour la même larve à la base.

Les polymorphismes en programmation
On distingue :

- Le **polymorphisme des traitements** (ou ad hoc) : Le même identifiant est utilisé pour désigner des séquences d'instructions différentes.
  ==> Mécanisme de surdéfinition des traitements/méthodes
- Le **polymorphisme des données** (ou universel) avec :
  - Le **polymorphisme d'héritage** : ce module.
  - Le **polymorphisme paramétrique** : cf. module @[Modèles].

Polymorphisme d'héritage
Dit aussi **polymorphisme de redéfinition** (*overriding*) ou encore **polymorphisme d'objets**, c'est le fait que les instances d'une sous-classe, en paramètres de fonctions/-procédures ou lors d'affectations, soient **substituables** aux instances des classes de leur ascendance.

Mise en oeuvre
Elle se fait par :

- Les mécanismes de l'**héritage** dans les hiérarchies de classes.
- La **liaison dynamique**.

1.2 Liaisons statiques et dynamiques

La liaison statique

Le premier aspect (déjà vu) est la possibilité de surcharger des modules (procédures et/ou fonctions) ou de redéfinir des méthodes à l'intérieur de classes dérivées. Dans le cas de la surcharge, c'est le nombre et le type des paramètres qui déterminent la version du module à exécuter. Lors de la redéfinition de méthodes, c'est le type de l'objet ou le type du pointeur sur l'objet qui indique la version à exécuter. Cette information est connue à la compilation. Un appel à un module ou une méthode est alors converti par le compilateur en un branchement à une adresse fixe, là où se trouve le code du module ou méthode : c'est une **liaison statique** (*static binding* en anglais).

La liaison dynamique

Dans le contexte de l'héritage, le deuxième aspect est celui qui permet de ne connaître, qu'au moment de l'exécution du programme, la version de la méthode à exécuter. On parle alors de **liaison dynamique** (*dynamic binding*) ou de **liaison différée** (*late binding*).

Exemple

Un exemple classique est le jeu d'échecs et ses pièces. Chaque pièce du jeu a un nombre plus ou moins limité de mouvements autorisés.

Une classe `Piece` va définir une méthode `mouvement`. Les classes dérivées `Pion`, `Fou`, `Tour`, `Cavalier`, `Roi` et `Reine` vont contenir aussi une méthode `mouvement` qui déterminera leurs mouvements propres.

Si on déclare un objet `o` de type `Piece` (classe de base) et qu'à un moment donné il réfère un objet de type `Tour`, alors si on accède à la méthode `mouvement`, c'est la méthode `Tour::mouvement` qui va être appelée.

Mise en oeuvre

Deux ingrédients :

- Les méthodes virtuelles, et
- Les références/pointeurs.

2 Méthode virtuelle

2.1 Déclaration d'une méthode virtuelle

Méthode virtuelle

Méthode déclarée **virtuelle** dans la classe de base et **redéfinie** dans les classes dérivées. Les versions redéfinies de cette méthode dans les classes dérivées doivent avoir **le même prototype** et sont considérées comme **implicitement virtuelles**.

C++ : Déclaration d'une méthode virtuelle

```
class B
{ ....
  virtual ... methode(liste_parametres) [const];
};
```

Explication

Déclare la `methode` comme étant virtuelle (mot-clé `virtual`). Celui-ci se place devant le prototype de la méthode, **uniquement** dans la déclaration de **la classe de base**.

2.2 Méthode virtuelle – Attention Piège

C++ : Mauvais code

```
class A { virtual void mv(...) ; };
class B : public A {
  virtual void mv(...) ; } ;
void g(A o) { o.mv(...) } //<- PARAMETRE PAR VALEUR
int main()
{
  A a(...);
  B b(...);
  g(a); // ici appel A::mv()
  g(b); // ici appel A::mv()
}
```

Explication

Le code est **NON** virtuel. En effet la procédure `g` prend son paramètre par **valeur** : elle va donc s'exécuter sur une **copie** de ce paramètre.

C++ : Mise en place du polymorphisme

Par défaut, une méthode **n'est pas polymorphe**. Afin de la rendre polymorphe, il faut respecter les trois principes :

- Déclarer la méthode virtuelle dans la classe mère à l'aide du modificateur `virtual` au devant de son prototype.
- La redéfinir en respectant scrupuleusement la même signature (même type de retour, même signature de paramètres, même caractère constant ou non) dans les classes dérivées.
- Utiliser la liaison dynamique. Ceci signifie qu'il faut passer l'objet par référence (ou pointeur) pour que la procédure `g` agisse sur l'instance d'origine.

C++ : Le bon code

```
class A { virtual void mv(...); };
class B : public A {
  virtual void mv(...); };
void g(A& o) { o.mv(...) } //<- PARAMETRE PAR REF
int main()
{
  A a(...);
  B b(...);
  g(a); // // ici appel A::mv()
  g(b); // ici appel B::mv()
}
```

Explication

Maintenant, la procédure `g` prend un paramètre par référence : cette fois tout fonctionne (comme souhaité).

2.3 Méthode virtuelle – Technique

Le but de cette technique est le suivant : le type de certains objets n'étant connu qu'à l'exécution, le mécanisme mis en place par les méthodes virtuelles permettra d'exécuter la version redéfinie d'une méthode correspondant au type de l'objet.

Dans ce cas, le polymorphisme est pleinement réalisé car lorsqu'un pointeur de type classe de base appelle une méthode virtuelle, c'est la version de la méthode définie dans la classe dérivée correspondant à l'objet pointé qui sera exécutée. Cette technique est particulièrement utile pour la généralisation d'un traitement.

Implémentation en C++
La liaison dynamique est implémentée en C++ à l'aide d'une table de méthode virtuelle, en fait une table de pointeurs de méthodes que le compilateur construit pour chaque classe utilisant des méthodes virtuelles. Chaque instance d'une classe contient un pointeur caché sur sa table. Lors d'une instruction, le compilateur déréférence, pour l'objet sur lequel pointe `p`, le pointeur correspondant dans sa table virtuelle par une opération d'indirection et ainsi la méthode appropriée sera appelée.

C++ : Recommandation
Une particularité du C++ recommande que toute classe possédant au moins une méthode virtuelle doit également avoir un destructeur virtuel. Ceci permet de s'assurer que le destructeur approprié sera exécuté.

Méthode virtuelle – Retenir
Lorsqu'une méthode **virtuelle** est invoquée à partir d'une **référence** ou d'un **pointeur** vers une instance, c'est la méthode associée au type réel de l'instance qui sera exécutée.

Structeurs et virtualité

- Un constructeur **ne** peut **pas** être virtuel.
- Il ne faut pas invoquer de méthode virtuelle dans un constructeur (l'objet n'étant pas encore construit).
- Il est conseillé de toujours définir les **destructeurs** comme étant virtuels (pour invoquer proprement tous les destructeurs des objets sous-jacents).
- L'opérateur d'affectation **ne** peut **pas** être virtuel.

Distinguer surcharge et polymorphisme
Dans le mécanisme de :

- Surcharge : c'est la **signature** qui différencie l'appel de deux méthodes de même nom.
- Polymorphisme : c'est le **type de l'objet** auquel est appliqué la méthode qui entraîne l'appel de la bonne variante de la méthode.

3 Collections hétérogènes

3.1 Collection hétérogène

Nous avons vu que :

- L'héritage et les méthodes virtuelles permettent de mettre en oeuvre des traitements génériques sur les instances d'une hiérarchie de classes (polymorphisme d'inclusion).
- Les méthodes génériques doivent utiliser des paramètres passés en référence pour que le traitement se fasse en fonction de la nature réelle de l'instance.

Mais qu'en est-il si un traitement donné doit porter sur un **ensemble d'instances** d'une hiérarchie de classe ?

C++ : Hiérarchie de classes

Considérons les déclarations suivantes :

```
class B { ... virtual void g(...) = 0 ; } ;
class D1 : public B { ... void g(...) ; } ;
class D2 : public B { ... void g(...) ; } ;
class Collection { ...
    protected: vector<B> v ;
} ;
```

Les instances contenues dans l'attribut `v` font partie d'une même hiérarchie de classe mais sont de nature hétérogène.

Mauvais codage

Un tel codage de la classe ne fonctionne pas (comme souhaité) :

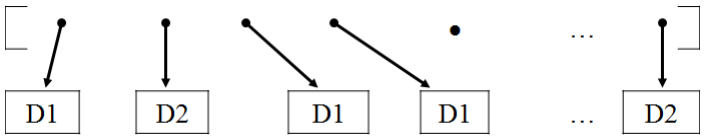
- L'attribut `v` est constitué d'instances de type `B` et **non pas de références** à ces instances.
- On ne peut pas mettre en oeuvre la liaison dynamique de la méthode `g` de la classe `B`.

La solution

Elle consiste à définir un conteneur de **pointeurs** (ou de références) :

```
vector<B*> v;
```

Ceci implique que seuls les pointeurs, c.-à-d. les **adresses** des instances, sont stockés dans le conteneur et non les instances elles-même.

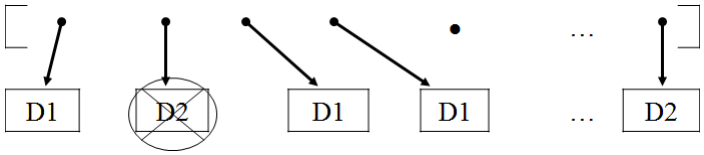


Conséquences

- Il devient possible de manipuler les instances originelles.
- Ceci réalise la **liaison dynamique**.

3.2 Pointeurs et intégrité des données

Cependant la classe `Collection` comporte un **danger potentiel** : pour que tout fonctionne bien (comme souhaité), il est nécessaire que les éléments pointés existent aussi longtemps que leurs pointeurs !



Mais comment(?) puisqu'une variable locale est détruite à la sortie du bloc qui la déclare ?

Les règles

Mais qui dit « allocation dynamique », dit « bonne gestion » et « programmation rigoureuse ». En particulier :

- Pensez, si nécessaire, à la copie profonde et au destructeur pour libérer la mémoire allouée.
- Attention ! En cas de copie profonde avec des classes abstraites (typiquement collections hétérogènes) il est nécessaire de définir une méthode de copie comme étant virtuelle pure au niveau de la classe abstraite.
- N'oubliez pas la règle d'or : c'est celui qui a alloué la mémoire (`new`) qui est chargé de la libérer (`delete`).

4 Compléments

4.1 Forme canonique de Coplien

Coplien suggère la représentation suivante pour toute redéfinition d'une méthode virtuelle :

1. Une séquence d'instructions nommée « PRE » à loger, dans la mesure du possible, dans une méthode séparée.
2. Le code de la classe `A`.
3. Une séquence d'instructions nommée « POST » à loger, dans la mesure du possible, dans une méthode séparée.

On pourra alors l'écrire ainsi :

```
class B::poly(typeParam1 param1, typeParam2 param2)
{
  PRE(paramètres spécifiques)
  A::poly(param1, param2);
  POST(paramètres spécifiques)
}
```

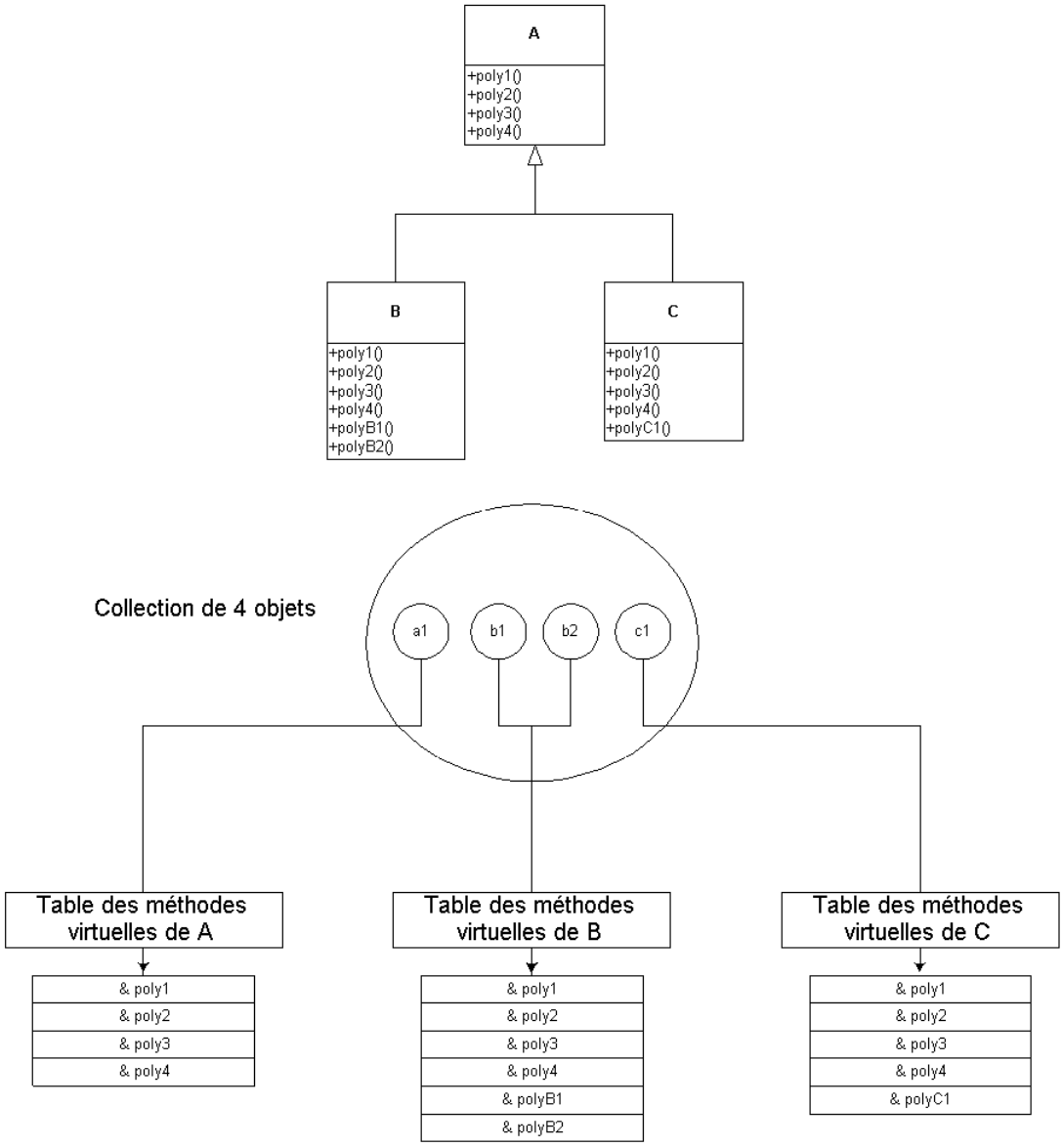
Ce formalisme s'appelle habituellement : forme canonique de Coplien des méthodes virtuelles.

Cette implémentation présente les nombreux avantages liés à la réutilisation de code, en particulier la fiabilisation de l'implémentation et l'évolutivité (une modification de la classe mère sera répercutée directement dans `B`).

4.2 Les tables de méthodes virtuelles

L'idée de base des tables de méthodes virtuelles est de toujours ranger les méthodes virtuelles dans le même ordre de manière à pouvoir les appeler par leur position dans la table.

Supposons, par exemple, que la classe `A` déclare quatre méthodes virtuelles respectivement nommées `poly1`, `poly2`, `poly3` et `poly4`. Ses sous classes `B` et `C` vont les redéfinir et éventuellement en rajouter. La figure suivante illustre ces trois classes, leurs tables de méthodes virtuelles obtenues ainsi que quatre objets avec leurs pointeurs vers les tables de méthodes virtuelles respectives :



Ainsi, l'appel à la méthode `poly2` ne s'effectue pas directement mais via la TVM. L'appel devient donc :

```
o->tvm_[1]();
```

Ainsi, cela permet de manipuler indirectement des entités regroupées dans un conteneur car l'appel aux méthodes virtuelles est sûr d'aboutir sur la bonne méthode.

5 C++ : Spécificités

6 Polymorphisme ([Drouillon-CC1 :c7], Juillet 2017)

6.1 Qu'est ce que le polymorphisme

En C++ est apparue la notion de typage dynamique également appelée « polymorphisme ». L'idée est qu'un pointeur puisse changer de type en fonction de l'objet sur lequel il pointe. Ceci signifie qu'il est possible avec un pointeur de reconnaître dynamiquement un objet afin d'accéder à ses membres. Cette propriété est introduite en C++ pour des méthodes redéfinies dans des classes dérivées. Elle permet d'introduire des listes d'objets de types différents dans les programmes. Typiquement c'est un tableau de pointeurs dans lequel chacun, non seulement pointe sur un objet différent, mais surtout peut sans typage (cast) accéder à certaines méthodes de cet objet. Ce changement important obéit à des règles strictes que nous allons explorer et vérifier.

6.2 Accès pointeur par défaut aux méthodes redéfinies

Soit une classe `A| et une classe \lstinline`B@ dérivée de `A`. Par défaut un pointeur de type `A*` auquel est affectée une adresse de type `B*` n'accède qu'aux données et méthodes de `A`.

Exemple

Le programme suivant permet de le constater :

```
#include <iostream>
using namespace std;

class A
{
public:
   int val;
   A(): val(0){}
   A(int v): val(v){}
   void afficher() const { cout<<"A : "<<val<<endl; }
};

class B:  public A
{
public:
   int val;
   B(int a,int b): A(a),val(b) {}
   void afficher() const { cout<<"B : "<<val<<endl; }
};

int main()
{
   A *ptr;
   B b(2,3);
   ptr = &b;
```

```
   ptr->afficher();
   return 0;
}
```

Bien que le pointeur a contienne l'adresse d'un objet B, le programme affiche :

```
A : 2
```

6.3 Accès pointeur aux méthodes virtuelles redéfinies

Méthodes virtuelles redéfinies

Le pointeur de type `A*` de l'exemple précédent peut accéder à des méthodes de `B` si :

- Ces méthodes sont des redéfinitions de méthodes de `A`.
- Les méthodes initiales de `A` sont déclarées virtuelles avec le qualificatif `virtual`.

Exemple

Voici le programme précédent modifié : la méthode `afficher()` de `A` est déclarée virtuelle :

```
#include <iostream>
using namespace std;

class A
{
public:
   int val;
   A(): val(0){}
   A(int v): val(v){}
   virtual void afficher() const { cout<<"A : "<<val<<endl; } //<-- MODIF
};

class B:  public A
{
public:
   int val;
   B(int a,int b): A(a),val(b) {}
   void afficher() const { cout<<"B : "<<val<<endl; } // virtual implicite
};

int main()
{
   A *ptr;
   B b(2,3);
   ptr = &b;
   ptr->afficher();
   return 0;
}
```

Le programme affiche :

```
B : 3
```

Avec une méthode déclarée virtuelle dans une classe, la détermination de la méthode effective, lors d'un appel via un pointeur d'objet, se fait à l'exécution en fonction du

type de l'objet dont le pointeur a l'adresse. Ici le type de l'objet dont le pointeur a contient l'adresse de `B` et comme la méthode `afficher()` de `A` est virtuelle, c'est la fonction `afficher()` de `B` qui est appelée au moment de l'exécution.

Destructeur virtuel

Cette notion de virtualité des méthodes peut s'appliquer à un destructeur afin de pouvoir libérer la mémoire de l'objet de l'objet réellement pointé par un pointeur.

```
#include <iostream>
using namespace std;

class A
{
public:
   int val;
   A(): val(0){}
   A(int v): val(v){}
   virtual void afficher() const { cout<<"A : "<<val<<endl; }
   virtual ~A() { cout<<"destructeur A"<<endl; } //<-- AJOUT
};

class B:  public A
{
public:
   int val;
   B(int a,int b): A(a),val(b) {}
   void afficher() const { cout<<"B : "<<val<<endl; } // virtual implicite
   ~B() { cout<<"destructeur B"<<endl; } //<-- AJOUT virtual implicite
};

int main()
{
   A *ptr;

   ptr = new A(1);
   ptr->afficher();
   delete ptr;

   ptr = new B(2,3);
   ptr->afficher();
   delete ptr;
}
```

Le programme affiche :

```
A : 1
destructeur A
B : 3
destructeur B
destructeur A
```

Destructeur virtuel : Règle

Bjarne Stroustrup, créateur du langage, souligne que s'il existe au moins une méthode virtuelle dans une classe, le destructeur DOIT être également VIRTUEL pour que lors de la destruction le bon destructeur soit également appelé.