Héritage multiple [hm]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

C++ - Héritage multiple

Mots-Clés Héritage multiple ■
Requis Classes, Relations entre classes, Héritage ■
Difficulté ●●○ (1 h) ■

Introduction

Ce module explore les arcanes de l'**héritage multiple**. On utilise cette technique pour représenter des relations du genre « A est un B **et aussi** un C ». Cette technique permet donc de créer une classe dérivée à partir de plusieurs classes de base.

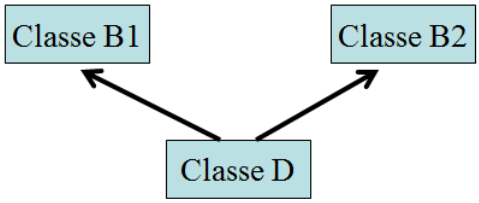
Remarque
Les langages JAVA et C# ne connaissent pas l'héritage multiple.

1 Héritage multiple

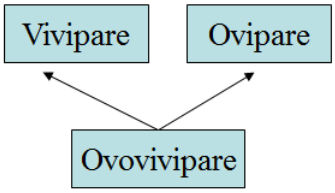
1.1 Héritage multiple

Héritage multiple
Sous-classe qui hérite de **plusieurs super-classes**.
Comme pour l'héritage simple, elle hérite des super-classes :

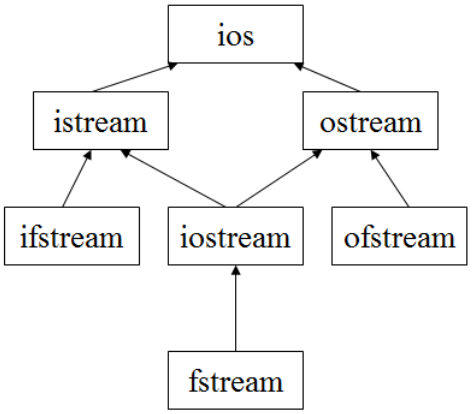
- Tous leurs attributs et méthodes (sauf les structeurs).
- Leur type.



Exemple : Zoologique



Exemple : Informatique



1.2 Déclaration d'un héritage multiple

Déclaration d'un héritage multiple

```
class D: public B1, public B2, ...
{ ... };
```

Explication

Définit la sous-classe `D` comme dérivant des super-classes `Bk`. Il n'y a pas de restriction sur le nombre de super-classes dont la sous-classe peut hériter.

Ordre de déclaration des super-classes

Il **induit** celui des **appels des constructeurs** des super-classes.

1.3 Constructeurs dans une hiérarchie multiple

C++ : Constructeurs dans une hiérarchie multiple

```
D::D(liste_arguments)
: B1(arguments1), ..., attribut1(valeur1), ...
{ ... }
```

Explication

Initialise les attributs hérités par invocation des constructeurs des super-classes (idem héritage simple).

Déclaration d'héritage

L'exécution des constructeurs des super-classes se fait selon l'ordre de la déclaration d'héritage, **et non pas** selon l'ordre des appels dans le constructeur.

Structeurs

- Lorsqu'une super-classe admet un constructeur par défaut, il n'est pas nécessaire de l'invoquer explicitement.
- L'ordre des appels des destructeurs des super-classes est l'inverse de celui des appels de constructeurs.

2 Ambiguïté des membres hérités

2.1 Accès direct ambigu – Problème

Une sous-classe peut **accéder directement** aux membres (attributs et méthodes) protégés de ses super-classes. Mais si ces membres portent le même nom dans plusieurs classes, il y a ambiguïté de l'appel.

Exemple : Accès direct ambigu

```
class Ovipare { public:
  void afficher() const; };
class Vivipare { public:
  void afficher() const; };
class Ovovivipare :
  public Ovipare,
  public Vivipare
  { ... };
int main()
{
  Ovovivipare o;
  o.afficher();
}
```

L'accès `o.afficher` provoque une erreur à la compilation même si la méthode `afficher` n'a pas les mêmes paramètres dans les deux classes `Ovipare` et `Vivipare`.

2.2 Accès direct ambigu – Solutions

Pour résoudre les ambiguïtés de noms des attributs/méthodes,

Solution naïve

Elle consiste à utiliser l'opérateur de résolution de portée :

```
int main()
{
  Ovovivipare o;
  o.Vivipare::afficher(); // A NE PAS FAIRE
}
```

Ici c'est l'utilisateur de la classe `Ovovivipare` qui doit décider du fonctionnement correct de cette classe. Alors que cette responsabilité doit normalement **incomber aux concepteurs** de la classe. **C'est donc une mauvaise solution**.

Instruction using

Une bonne solution consiste à lever l'ambiguïté en indiquant **explicitement** (au niveau de la conception) quelle(s) méthode(s) on souhaite invoquer :

```
class Ovovivipare : public Ovipare, public Vivipare { public:
  using Vivipare::afficher
};
```

Masquage de la méthode

Une autre bonne solution consiste à masquer la méthode (ayant créée l')ambiguïté en incorporant dans la sous-classe une méthode définissant la bonne interprétation de l'invocation ambiguë :

```
class Ovovivipare : public Ovipare, public Vivipare { public:
  void afficher() const
  {
    Ovipare::afficher();
    cout << " mais aussi...";
    Vivipare::afficher();
  }
};
```

2.3 Instruction using

Instruction using

```
using NomSuperClasse::NomMembreAmbigu;
```

Explication

Ajoute à la liste des déclarations des méthodes/attributs de la sous-classe, une déclaration spéciale indiquant quel(s) membre(s) (attribut/méthode) seront référencé(s) exactement.

Attention

Aucune parenthèses (ni prototypes) derrière les noms de méthodes dans un `using`.

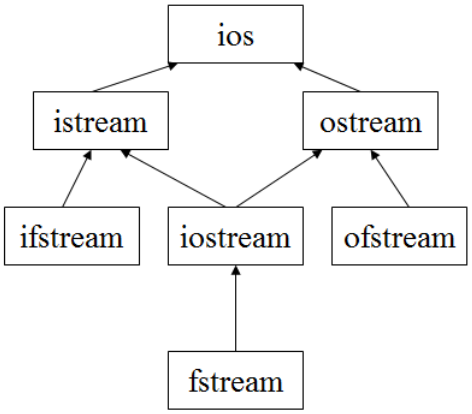
3 Héritage virtuel

3.1 Héritage virtuel

Héritage en diamant

Il peut se produire qu'une super-classe soit incluse plusieurs fois dans une hiérarchie à héritage multiple : on parle alors d'**héritage en diamant**, appelé aussi **héritage à répétition** (par certains auteurs).

Exemple : Informatique



Conséquence

Par définition, les membres (attributs/méthodes) de la super-classe seront inclus plusieurs fois. Dans l'exemple, chaque objet de la classe `iostream` (ou `fstream`) possèdera deux copies des attributs de la classe `ios`.

Classe virtuelle

Pour éviter la duplication des attributs d'une super-classe (donc obtenir un seul héritage), on peut utiliser le mécanisme de la dérivation virtuelle. Cette super-classe sera alors dite **virtuelle** (à ne pas confondre avec classe abstraite !).

3.2 Déclaration d'un héritage virtuel

Déclaration d'un héritage virtuel

```
class D: public virtual B
{ ... };
```

Explication

Déclare comme étant virtuel le lien d'héritage avec toutes ses sous-classes. L'ordre des modificateurs `virtual` et `public` dans un héritage virtuel est sans conséquence.

Déclaration au niveau 1

La propriété de virtualité est une caractéristique de la dérivation et non de la classe virtuelle. C'est pourquoi la déclaration de la virtualité (mot-clé `virtual`) s'effectue au niveau 1 de l'héritage (c.-à-d. super-classe). Ensuite c'est trop tard.

Remarque

Une classe dérivée peut avoir des classes de base à la fois virtuelles et non virtuelles.

Gestion des appels aux constructeurs

Dans un héritage virtuel, elle s'effectue comme suit :

- Si le constructeur d'un objet de la classe la plus dérivée est invoqué : les appels explicites au constructeur de la super-classe virtuelle dans les classes intermédiaires sont ignorés.
- Si la super-classe virtuelle a un constructeur par défaut : il n'est pas nécessaire de faire appel à ce constructeur explicitement dans sa classe la plus dérivée.
- Si l'appel explicite au constructeur de la super-classe virtuelle est omis de la sous-classe la plus dérivée et si cette super-classe virtuelle n'a pas de constructeur par défaut : le compilateur signale une erreur.

3.3 Exemples : Héritage virtuel et Constructeurs

Héritage virtuel (Informatique)

```
class ios { ... };
class istream : public virtual ios { ... }; //<- héritage virtuel
class ostream : public virtual ios { ... }; //<- héritage virtuel
class iostream : public istream, public ostream { ... };
```

Un seul objet de la super-classe `ios` est hérité par l'héritage commun des sous-classes `istream` et `ostream`.

Constructeurs (Zoologique)

```
class Animal { ... };
class Ovipare : public virtual Animal { ... };
class Vivipare : public virtual Animal { ... };
class Ovovivipare : public Ovipare, public Vivipare { public:
   Ovovivipare(...)
   : Animal(...), Ovipare(...), Vivipare(...), ...
   { ... }
};
```

- Ici les appels explicites au constructeur de la classe `Animal` dans les constructeurs de `Ovipare` et `Vivipare` sont ignorés.
- Si `Animal` dispose d'un constructeur par défaut : il n'est pas nécessaire de l'invoquer explicitement dans le constructeur de `Ovovivipare`.
- Si `Ovovivipare` ne fait pas un appel explicite au constructeur de `Animal` et si `Animal` n'a pas de constructeur par défaut : le compilateur signale une erreur.

3.4 Expérience sur l'héritage virtuel

Mise en oeuvre

L'exemple suivant montre comment utiliser l'héritage virtuel pour résoudre les problèmes liés à l'héritage à répétition :

```
// Mise en place de l'héritage virtuel
// Classe de base du système
class A
{
  public:
    A ()
    {
      cout << "Constructeur de A" << endl;
    }
  private:
    int a;
};
// Sous classe directe de A
class B: virtual public  A
{
  public:
    // Le constructeur de B appelle celui de A
    // afin d'initialiser correctement les attributs de A
    // présents dans la classe B


    B(): A()
    {
      cout << "Constructeur de B" << endl;
    };
  private:
    int b;
};
// Sous classe directe de A
class C: virtual public A
{
  public:
    // Le constructeur de C appele celui de A
    // afin d'initialiser correctement les attributs de A
    // présents dans la classe C
    C(): A ()
    {
      cout << "Constructeur de C" << endl;
    };
  private:
    int c;
};
// Sous classe de B et C
// Afin de gérer l'héritage à répétition, on introduit également A dans la liste
// des super classes
class D: virtual public A, public B, public C
{
  public:
    D(): A (),
         B (),
         C ()
```

```
    {
      cout << "Constructeur de D" << endl;
    };
  private:
    int d;
};
```

En plus de l'héritage double sur les classes `B` et `C`, ce code ajoute explicitement la classe `A` en super classe de `D`. En outre, `A| doit arriver en première position dans la liste des super classes et l'héritage sur \lstinline`A@ doit de nouveau être virtuel. Cette construction garantit que :

1. Les attributs de `A` ne seront présents qu'en un seul exemplaire, et ce, directement depuis `A`
2. Le constructeur de `A` est explicitement appelé dans celui de `D` assurant ainsi l'initialisation correcte des attributs hérités de `A`. Conséquemment, les appels au constructeur de `A` depuis ceux de `B` et `C` ne seront pas exécutés.

Remarque : L'ordre des modificateurs `virtual` et `public` dans un héritage virtuel est sans conséquence.

Vous voulez une preuve ? Allez, je suis bon prince, en voici une, minimaliste mais suffisante. Le code d'utilisation des classes précédentes est le suivant :

```
//Essai de l'héritage virtuel
int main(int, char **)
{
  B b1;
  cout << endl;
  C c1;
  cout << endl;
  D d1;
  return 0;
}
```

Et voici le résultat commenté :

```
Constructeur de A  // Le constructeur de B (pour l'objet b1) appelle bien
   celui de A
Constructeur de B
Constructeur de A  // Le constructeur de C (pour l'objet c1) appelle bien
   celui de A
Constructeur de C
Constructeur de A  // Le constructeur de D appelle une fois le
   constructeur de A
Constructeur de B  // une fois le constructeur de B qui n'appelle celui
   de A du fait de l'héritage virtuel
Constructeur de C  // une fois le constructeur de C qui n'appelle celui
   de A du fait de l'héritage virtuel
Constructeur de D  // puis finalement son code propre !
```

Quand je vous le disais que le constructeur de `A` ne serait appelé qu'une seule fois ! Les attributs placés dans les classes ne sont pas mis à contribution par cet exemple. En revanche, ils deviendront d'une importance cruciale dans l'expérience proposée à la fin de cette section.

Conséquences néfastes de l'héritage virtuel

Tout ceci semble trop parfait pour être honnête, et, bien entendu, il va falloir en payer les conséquences. Tout d'abord, il vous faut savoir qu'il est absolument interdit de faire du transtypage descendant à travers un héritage virtuel. Par exemple, le code suivant est illégal :

```
A *p;
B *q;
...
q=(B *)a;
```

A noté il est rarement légitime de faire des transtypages descendants, et si vous en avez besoin, c'est qu'il y a probablement des incohérences dans votre modèle objet.

En outre, et sans vouloir rentrer dans les détails internes de codage du C++ (voir à ce sujet l'ouvrage « Le modèle orienté objet du C++ » de l'inévitable et excellent Stanley Lippman), l'héritage virtuel alourdit considérablement la gestion de la table des méthodes virtuelles et des attributs, il faut donc l'utiliser uniquement à bon escient.

Quand devez vous imposer de l'héritage virtuel ?

Supposons que vous fournissiez une bibliothèque dont toutes les classes dérivent d'un ancêtre commun. Est-il nécessaire d'utiliser de l'héritage virtuel afin qu'un de vos clients puisse faire de l'héritage multiple sans danger ?

Tout d'abord, il vous faut décider si vos classes ont une chance d'être dérivées par votre client et si oui, peut-il désirer les associer à une autre classe de votre bibliothèque ? si la réponse est oui, alors vous devez prévoir de l'héritage virtuel, dans tous les autres cas, utilisez de l'héritage « normal ».

Expérience sur l'héritage virtuel

Dans le code précédent, retirez chacun à son tour les modificateurs virtuels dans l'héritage ou l'héritage direct sur `A` dans la classe `D` afin de voir quels messages sont générés par votre compilateur préféré. Par exemple, l'oubli du mot clé `virtual` dans la dérivation de `C` depuis `A` provoque respectivement le message de compilation (avec gcc) et l'affichage d'exécution suivants :

A la compilation :

```
essvirt.cc:62: warning: direct base ‘A’ inaccessible in ‘D’ due to
   ambiguity
```

A l'exécution :

```
Constructeur de A  // Le constructeur de B (pour l’objet b1) appelle bien
   celui de A
Constructeur de B
Constructeur de A  // Le constructeur de C (pour l’objet c1) appelle bien
   celui de A
Constructeur de C
Constructeur de A  // Le constructeur de D appelle une fois le
   constructeur de A
```

```
Constructeur de B  // une fois le constructeur de B qui n'appelle celui
   de A du fait de l'héritage virtuel
Constructeur de A  // comme C n'hérite pas virtuellement de A, le
   constructeur de C appelle celui de A
                   // sur sa copie des attributs de A
Constructeur de C  // appel du constructeur de C
Constructeur de D  // puis finalement son code propre !
```

Le message de compilation de gcc étant clairement du à la duplication des attributs de `A` dans la classe `D`.

4 C++ : Héritage multiple ([Drouillon-CC1 :c7], Juillet 2017)

4.1 Héritage multiple

Qu'est ce que l'héritage multiple

Une classe peut hériter de plusieurs classes simultanément. Il suffit de le préciser à la définition de la classe dérivée :

```
class A1 {...};
class A2 {...};
class B : [public ou protected ou private] A1, [public ou ...] A2{
...
};
```

Tout ce qui a été vu de l'héritage est vrai pour l'héritage multiple à deux précisions près :

- Les constructions.
- Les relations transversales.

Constructions dans un héritage multiple

Dans la classe dérivée, l'ordre d'appel des constructeurs des classes de base est donné par l'ordre des déclarations des classes de base.

Exemple

Soit une classe `C| dérivée des classes \lstinline`A1@ et `B2`, avec `B2` elle-même dérivée de `A2`. Chaque classe comporte un attribut `val` et un constructeur affichant le nom de sa classe et son attribut.

```
#include <iostream>
using namespace std;
class A1
{
public:
   int val;
   A1(int v)
   : val(v)
   { cout<<"A1 : "<<val<<endl; }
};

class A2
{
public:
   int val;
   A2(int v)
   : val(v)
   { cout<<"A2 : "<<val<<endl; }
};

class B2: public A2
{
public:
   int val;
```

```
   B2(int bv, int av)
   : A2(av), val(bv)
   { cout<<"B2 : "<<val<<endl; }
};

class C: public A1, public B2
{
public:
   int val;
   C(int a1v, int a2v, int bv, int cv)
   : B2(a2v, bv), A1(a1v), val(cv)
   { cout<<"C : "<<val<<endl; }
};

int main()
{
   C c(1,2,3,4);
}
```

Le programme affiche :

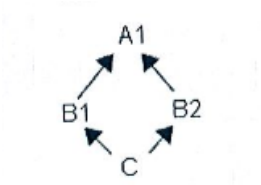
```
A1 : 1
A2 : 3
B2 : 2
C : 4
```

Relations transversales

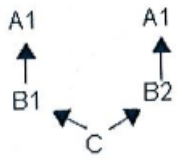
L'héritage multiple peut permettre de créer des relations transversales dans un arbre de classes. Toutefois il faut veiller à ne pas s'y perdre sachant qu'il peut devenir complexe à manipuler et qu'un certain nombre d'ambiguïtés peuvent être difficiles à repérer.

Exemple

Soient une classe de base `A1` et deux classes dérivées `B1` et `B2`, puis une classe `C` dérivée de `B1` et `B2`. Cela donne :



Lorsque nous déclarons un objet de la classe `C`, combien de fils contient-il des données de la classe `A1` : une ou deux fois ? La réponse est deux. Le schéma est correct pour le modèle mais en mémoire cela donne :



Pour tester :

```
#include <iostream>
using namespace std;
class A1
{
public:
   int val;
   A1(int v): val(v) {}
   void afficher() const { cout<<"A1 : "<<val<<endl; }
};

class B1: public A1
{
public:
   int val;
   B1(int a,int b): A1(a), val(b) {}
   void afficher() const
   {
      A1::afficher();
      cout<<"B1 : "<<val<<endl;
   }
};

class B2: public A1
{
public:
   int val;
   B2(int a, int b): A1(a), val(b) {}
   void afficher() const
   {
      A1::afficher();
      cout<<"B2 : "<<val<<endl;
   }
};

class C: public B1, public B2
{
public:
   int val;
   C(int a1, int b1, int a2, int b2, int c)
   : B1(a1, b1), B2(a2, b2), val(c)
   {}

   void afficher() const
   {
      B1::afficher();
      B2::afficher();
      cout<<"C : "<<val<<endl;
   }
};

int main()
{
   C c(1,2,3,4,5);
   c.afficher();
}
```

Le programme affiche :

```
A1 : 1
B1 : 2
A1 : 3
B2 : 4
C : 5
```

Pour éviter ce fonctionnement et n'avoir qu'une seule fois les données et méthodes de la classe de `A1`, il faut utiliser la notion de virtualité et réaliser un héritage virtuel.

4.2 Héritage multiple avec une base virtuelle

Héritage multiple, virtualité

Dans un héritage multiple il est possible de n'avoir qu'une fois les données de la classe de base en mémoire avec le qualificatif `virtual` dans les classes dérivées :

```
class A {...}; // base
class B1 : virtual public A {...};
class B2 : virtual public A {...};
class C : public B1, public B2 {...};
```

La règle est que l'ensemble des héritages déclarés `virtual` d'une même classe de base constitue une seule fois les attributs de cette classe de base en mémoire. De plus, il convient d'avoir un constructeur par défaut dans la classe de base, sous peine de générer une erreur :

```
no matching function to call 'A::A()'
```

Exemple : Héritage virtuel (1)

Le programme suivant permet de le constater :

```
#include <iostream>
using namespace std;
class A1
{
public:
   int val;
   A1(): val(0) {} //<-- AJOUT
   A1(int v): val(v) {}
   void afficher() const { cout<<"A1 : "<<val<<endl; }
};

class B1: virtual public A1
{
public:
   int val;
   B1(int a,int b): A1(a), val(b) {}
   void afficher() const
   {
      A1::afficher();
      cout<<"B1 : "<<val<<endl;
   }
};
```

```
class B2: virtual public A1
{
public:
   int val;
   B2(int a, int b): A1(a), val(b) {}
   void afficher() const
   {
      A1::afficher();
      cout<<"B2 : "<<val<<endl;
   }
};

class C : public B1, public B2
{
public:
   int val;
   C(int a1, int b1, int a2, int b2, int c)
   : B1(a1, b1), B2(a2, b2), val(c)
   {}

   void afficher() const
   {
      B1::afficher();
      B2::afficher();
      cout<<"C : "<<val<<endl;
   }
};

int main()
{
   C c(1,2,3,4,5);
   c.afficher();
   cout<<"-----------"<<endl;
   c.A1::val = 9;
   c.afficher();
}
```

Le programme affiche :

```
A1 : 0
B1 : 2
A1 : 0
B2 : 4
C : 5
-----
A1 : 9
B1 : 2
A1 : 9
B2 : 4
C : 5
```

Le deuxième affichage montre bien, avec une affectation de la valeur 9, que la variable `val` de la base est bien unique. Mais le premier affichage met en évidence un problème avec les constructeurs : le compilateur ne sait pas choisir entre l'appel du constructeur de `A` en `B1` ou en `B2`. Tel que nous avons écrit le programme, les appels constructeurs de

A en B1| et \lstinlineB2@ sont ignorés. En fait c'est le constructeur par défaut de A| qui est appelé en \lstinlineC@. En effet, dans le cas où une classe petite fille hérite de plusieurs classes filles déclarées virtual, C++ autorise et nécessite un appel constructeur de la classe mère (base) dans la classe petite fille.

Exemple : Héritage virtuel (2)

Voici le programme avec des constructeurs adaptés :

```
#include <iostream>
using namespace std;
class A1
{
public:
   int val;
   A1(): val(0) {}
   A1(int v): val(v) {}
   void afficher() const { cout<<"A1 : "<<val<<endl; }
};

class B1: virtual public A1
{
public:
   int val;
   B1(int b): val(b) {} //<-- AJOUT
   B1(int a,int b): A1(a), val(b) {}
   void afficher() const
   {
      //A1::afficher(); //<-- SUPPRESSION
      cout<<"B1 : "<<val<<endl;
   }
};

class B2: virtual public A1
{
public:
   int val;
   B2(int b): val(b) {} //<-- AJOUT
   B2(int a, int b): A1(a), val(b) {}
   void afficher() const
   {
      //A1::afficher(); //<-- SUPPRESSION
      cout<<"B2 : "<<val<<endl;
   }
};

class C : public B1, public B2
{
public:
   int val;
   C(int a1, int b1, int b2, int c) //<-- MODIF
   : A1(a1), B1(b1), B2(b2), val(c) //<-- MODIF
   {}

   void afficher() const
   {
      A1::afficher(); //<-- AJOUT
      B1::afficher();
```

```
      B2::afficher();
      cout<<"C : "<<val<<endl;
   }
};

int main()
{
   C c(1,2,3,4); //<-- MODIF
   c.afficher();
}
```

Le programme affiche :

```
A1 : 1
B1 : 2
B2 : 3
C : 4
```