Université de Lyon – Université Claude Bernard Lyon 1
Master d'Informatique

Programmation Avancée Les différents mécanismes des langages et de C++ pour la généricité

Norme ISO

Raphaëlle Chaine
raphaelle.chaine@liris.cnrs.fr
2020-2021

1

Héritage multiple

- Conflits d'ambiguïtés possibles
  - héritage d'identificateurs identiques provenant de bases différentes

```
class Programmeur
{...
  int niveau_qualification;
};

class Analyste
{...
  int niveau_qualification;
};

class Analyste_Programmeur : public Analyste,
                             public Programmeur
{...};

Analyste_Programmeur ap;
std::cout << ap.niveau_qualification; //NON Ambigu
```

2

2

- Recours à l'opérateur de résolution de portée pour lever les ambiguïtés

```
Analyste_Programmeur ap;
std::cout << "Niveau de qualification en temps qu'analyste "
          << ap.Analyste::niveau_qualification
          << " \nNiveau de qualification en temps que Programmeur "
          << ap.Programmeur::niveau_qualification;
```

3

3

- Le masquage supprime les ambiguïtés

```
class Programmeur
{...
  int niveau_qualification;
};

class Analyste
{...
  int niveau_qualification;
};

class Analyste_Programmeur : public Analyste,
                             public Programmeur
{...
 int niveau_qualification;
};

Analyste_Programmeur ap;
std::cout << ap.niveau_qualification
          << ap.Analyste::niveau_qualification
          << ap.Programmeur::niveau_qualification;
```

4

4

- La résolution de surcharge ne s'étend pas au delà de la portée d'une classe

```
class Programmeur
{...
  void presentation();
};

class Analyste
{...
  void presentation(int);
};

class Analyste_Programmeur : public Analyste,
                             public Programmeur
{...};

Analyste_Programmeur ap;
ap.presentation(); //Ambigu !!!!
ap.presentation(3); //Ambigu !!!!
ap.Programmeur::presentation();
ap.Analyste::presentation(5);
```

La résolution de surcharge ne peut se faire qu'entre fonctions d'une même portée!

5

5

- utiliser une using-declaration pour ramener les surcharges dans une même portée

```
class Programmeur
{...
  void presentation();
};

class Analyste
{...
  void presentation(int);
};

class Analyste_Programmeur : public Analyste,
                             public Programmeur
{...
 using Programmeur::presentation;
 using Analyste::presentation;
};

Analyste_Programmeur ap;
ap.presentation();
ap.presentation(3);
```

6

6

- Exo :

```
class Analyste          class Programmeur
{public :               {public :
  int f(char);            int f(double);
  int f(int);             int f(int);
 };                      };
class AnalysteProgrammeur : public Analyste,
                            public Programmeur
{public :
  using Analyste::f; using Programmeur::f;
  int f(int);
};
```

Quelles sont les versions des fonctions appelées depuis g()?
Y a t'il des conflits d'ambiguïté?

```
void g(AnalysteProgrammeur &d)
{ d.f('a');
  d.f(5.9);
  d.f(1);
}
```


- Solution

```
void g(AnalysteProgrammeur &d)
{ d.f('a');       // Analyste::f('a')
  d.f(5.9);       //Programmeur::f(5.9)
  d.f(1);         //AnalysteProgrammeur::f(1)
}
```

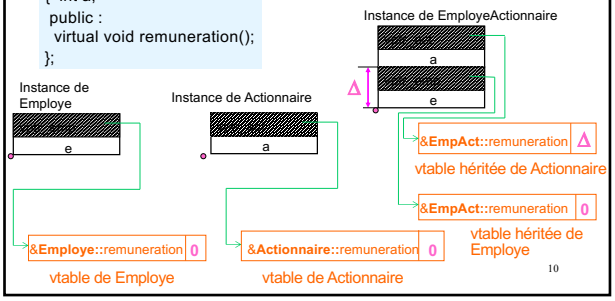

Héritage multiple et classes polymorphes

- Classe dérivée de **plusieurs** classes polymorphes :
  - Autant de pointeurs supplémentaires vers des tables virtuelles
- Exemple :
  - Classe *EmployeActionnaire* spécialisation des classes polymorphes *Employe* et *Actionnaire*
  - Encombrement supplémentaire des instances de *EmployeActionnaire :*
    - Un pointeur vers la table virtuelle héritée de Employe
    - Un pointeur vers la table virtuelle héritée de Actionnaire
- **La table virtuelle doit également fournir le décalage pour positionner *this* sur l'objet effectivement pointé …**


```
class Employe
{  int e;
 public :
  virtual void remuneration();
};
```

```
class Actionnaire
{ int a;
 public :
  virtual void remuneration();
};
```

```
class EmployeActionnaire :
  public Employe,
  public Actionnaire
{ int ea;
 public :
  void remuneration();
};
```



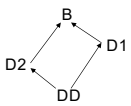
- EmployeActionnaire gerard;
  Employe & emp=gerard;
  Actionnaire & act=gerard;

  gerard.remuneration(); // Après avoir décalé this de 0 ,
  // Appel deEmpAct::remuneration

  emp.remuneration(); // Après avoir décalé this de 0 ,
  // Appel de EmpAct::remuneration

  act.remuneration(); // Après avoir décalé this de Δ ,
  // Appel de EmpAct::remuneration

- Classe de base dupliquée
  - Une classe dérivée **ne peut avoir 2 bases directes identiques**
  - Cependant, une même classe peut être héritée **plusieurs fois** à travers des chemins différents d'héritage indirect
  - une telle classe est dupliquée dans les instances de la classe dérivée



```
class lecteur_bande          class tuner          class moniteur
{ ... };                     { ...                { ... };
                               int canal;
                             };
class magnetoscope : public lecteur_bande, public tuner
{ ... };

class televiseur : public moniteur, public tuner
{... };

class teletoscope : public magnetoscope,
                    public televiseur
{ ... };
```

Pierre dit que c'est un "combiné"

- Avec ce schéma d'héritage, un teletoscope* se retrouve implanté avec un **double tuner**

```
teletoscope tt;              tt.magnetoscope::canal=5;
tt.canal=50; //ambigu        tt.televiseur::canal=5000;
```

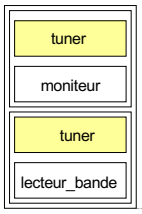
* ici utilisé pour "téléviseur avec magnétoscope intégré"

13

13

- … mais un televiseur avec magnetoscope integré pas cher **ne dispose probablement que d'un tuner !**
- Le schéma d'héritage utilisé pose alors des problèmes :
  - de taille mémoire gaspillée
  - de cohérence entre les informations dupliquées

Teletoscope tt;



&tt

14

14

- Problème supplémentaire :
  La possibilité de conversion standard (*upcast*) d'un teletoscope en tuner devient ambiguë

```
teletoscope tt;
tuner * tu;
tu =&tt; // ambigu, quel tuner?
```

- Il faut d'abord caster le teletoscope* en magnetoscope* ou en televiseur*, puis en tuner *

```
tu = static_cast<magnetoscope *> &tt; //OK
tu = static_cast<televiseur * > &tt; //OK
```

15

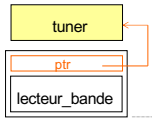
15

Héritage virtuel

- La notion de base virtuelle permet d'éviter le problème de la duplication d'une classe de base
- Syntaxe :

```
class magnetoscope : public lecteur_bande,
                     virtual public tuner
{...
};

magnetoscope m;
```



&m

16

16

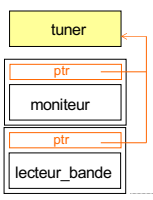
- Si la classe tuner est également une base virtuelle de la classe televiseur

```
class televiseur : public moniteur,
                   virtual public tuner {...};
```

- Un teletoscope est alors constitué d'un unique tuner*

```
class teletoscope : public magnetoscope,
                    public televiseur {...};
```

```
teletoscope tt;
tt.canal=300;
tuner & tu=tt;
```



&tt

*Une base virtuelle existe à un seul exemplaire dans les instances de ses descendantes

17

17

Héritage virtuel et constructeurs

- On suppose les classes précédentes munies des constructeurs suivant :

```
class tuner
{ …
  int canal;
  tuner(int c=500) : canal(c) {}
};

class magnetoscope : public lecteur_bande,
                     virtual public tuner
{ …
  magnetoscope() : tuner(100) {}
};

class televiseur : public moniteur,
                   virtual public tuner
{…
  televiseur() : tuner(300) {}
};

class teletoscope : public magnetoscope,
                    public televiseur
{ …
  teletoscope() : magnetoscope(),televiseur(){}
};

teletoscope tt;
```

Quelle est la valeur du canal de tt ?

18

18

- Réponse : **500**
- Les sous-objets provenant de classes de bases virtuelles indirectes sont initialisés :
  - par appel à **leur constructeur** par défaut
  - ou par appel à un autre constructeur de la base virtuelle, si mentionné dans la liste d'initialisation

```
class teletoscope : public magnetoscope, public televiseur
{ …
  teletoscope();
};
```

Cas 1

```
teletoscope:: teletoscope() : magnetoscope(),televiseur(){}
  //equivalent à : tuner(), magnetoscope(), televiseur() {}
teletoscope tt;    // tt.canal vaut 500
```

Cas 2

```
teletoscope:: teletoscope() : tuner(700), magnetoscope(), televiseur() {}
teletoscope tt;    // tt.canal vaut 700
```

19

19

- Une base virtuelle indirecte est toujours construite avant les sous-objets auxquels elle est commune
- Mais attention !

```
class tuner
{ …
  int canal;
  tuner(int c=500) : canal(c) {}
};

class televiseur : public moniteur,
                   virtual public tuner
{…
  televiseur() : tuner(300) {canal=3;}
};

class magnetoscope : public lecteur_bande,
                     virtual public tuner
{ …
  magnetoscope() : tuner(100) {canal=1;}
};

class teletoscope : public magnetoscope,
                    public televiseur
{ …
  teletoscope() : tuner(700),
                  magnetoscope(),
                  televiseur(){}
};

teletoscope tt;
```

Quelle est la valeur du canal de tt ?

20

20

Héritage virtuel et protection d'accès

- Si une base virtuelle est héritée plusieurs fois, avec des modes d'accès différents
- L'accès le moins restrictif est dominant

```
class tuner
{ …
  public :
    void affiche {…}
};

class televiseur : public moniteur,
                   virtual public tuner
{ … };

class magnetoscope : public lecteur_bande,
                     virtual private tuner
{ … };

class teletoscope : public magnetoscope,
                    public televiseur
{ … };

teletoscope tt;
tt.affiche(); //OUI, accès à l'interface de tuner
```

21

21

Héritage virtuel et conversions

- Conversion standard d'une classe dérivée vers une classe de base virtuelle public (*upcast*)

```
tuner * tu = new teletoscope; // OK
```

- Pas de conversion d'une classe de base virtuelle vers une classe dérivée (*downcast*) avec l'opérateur static_cast

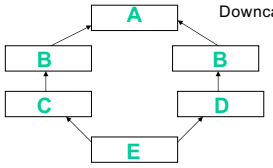
```
teletoscope *t=static_cast<teletoscope *>tu; //NON
        t= (teletoscope *) tu; //NON
```

- Possible en revanche avec dynamic_cast<> et RTTI (sauf ambiguité)

22

22

- Héritage multiple, virtuel, conversions et ambiguités …
  - *Upcast* :
    Ambiguité si upcast vers une classe de base plusieurs fois ancêtre et non virtuelle
  - *Downcast* depuis une base virtuelle polymorphe *:*



Downcast ambigu d'un A* en B*

23

23