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

Programmation Avancée
Les différents mécanismes des langages (dont C++) pour la généricité

Norme ISO

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

Redéfinitions de fonctions membres et polymorphisme

Intérêt du polymorphisme

- Programmer des classes avec des **fonctionnalités générales**, sans rentrer dans les détails liés aux différents cas de figures
  ex : fonctionnalités d'un moteur
- Programmer les **détails des fonctionnalités** dans des classes dérivées
  ex: fonctionnement d'un moteur à essence
  … ou diesel
- **Utiliser** les fonctionnalités spécialisées d'un objet référencé **sans connaître** de **quelle spécialisation** il s'agit

En JAVA

- Syntaxe de l'héritage et de la redéfinition de méthode en Java :

```
class Cadre extends Employe
{ …
  public void affiche()
   { super.affiche(); //affiche de la classe Employe
     system.out.println(" Cadre ");
   }
} // Mot clé super pour accéder à la méthode de
  //la classe mère
Employe gege = new Cadre;
gege.affiche();
```


En JAVA

- Syntaxe de l'héritage et de la redéfinition de méthode en Java :

```
Employe gege = new Cadre;
gege.affiche();
```

Mise en œuvre du polymorphisme : c'est la méthode affiche redéfinie dans Cadre qui est appelée!

Redéfinition de fonctions membres

```
class Employe
{ …
  int f(double);
};
```

```
class Cadre : public Employe
{ …
  int f(double); //redéfinition
};
```

- Si on souhaite qu'une instance de la classe Cadre réponde à une requête :
  - qu'on peut soumettre à un Employe,
  - mais de manière améliorée, ou simplifiée

  il convient de redéfinir la fonction membre f correspondante
- Déclaration de la version enrichie ou simplifiée de f dans la définition de Cadre

Une fonction membre de la classe Cadre est une redéfinition d'une fonction membre de la classe Employe **si** elles partagent un même prototype

- Remarque :
  La seule différence autorisée peut intervenir sur le type de la valeur de retour (covariance limitée)
  - si Employe::f retourne un TB* (resp TB &)
  - si Cadre::f retourne un TD* (resp TD &)

  et que TB est une **base public** de TD
- Attention :
  Si les fonctions partagent juste le même nom, il ne s'agit plus d'une redéfinition!

107

107

- Lors d'un appel, c'est le type statique de l'instance appelante qui détermine la version utilisée (résolution statique à la compilation)

```
Cadre c; Employe e;
Cadre *ac; Employe *ae;

e.f();
c.f();
e=c; e.f();
ae=&c; ae->f();
ac=static_cast<Cadre*>(ae); ac->f();
```

108

108

- Lors d'un appel, c'est le type statique de l'instance appelante qui détermine la version utilisée (résolution statique à la compilation)

```
Cadre c; Employe e;
Cadre *ac; Employe *ae;
e.f(); //Appel à Employe::f
c.f(); //Appel à Cadre::f
e=c; e.f(); //Appel à Employe::f
ae=&d; ae->f(); //Appel à Employe::f
ac=static_cast<Cadre*>(ae); ac->f(); //Appel à Cadre::f
```

- LE POLYMORPHISME N'EST DONC PAS MIS EN ŒUVRE!
  - POURQUOI?
  - EST-CE QUE CA PRESENTE UN INTERET?

109

109

- On peut néanmoins invoquer la version f de la base Employe sur une instance de la classe dérivée Cadre (opérateur de résolution de portée)

  d.Employe**::**f(); (équivalent du super de Java)

110

110

Principe du masquage

- Le mécanisme de redéfinition de fonction repose sur un simple principe de masquage :
  - la définition d'une classe dérivée D de B correspond à l'introduction d'une nouvelle **portée** (qui prévaut localement sur celle de B)
  - l'entrée de l'identificateur f dans la portée de D, **masque tout identificateur identique** hérité de B

111

111

```
class B
{public :
   int f(int);
};

class D : public B
{public :
   int f(int); //Redefinition
};

D d;
d.f(3);
```

Et là?

```
class B
{public :
   int f(int);
};

class D : public B
{public :
   int f(int,int); //Surcharge
};

D d;
d.f(3); ??
```

112

112

- AAAARGGGGHHH! Problème du masquage :
  - Le masquage ne se base que sur les identificateurs, **pas sur leur type**
  - Les identificateurs f définis ou déclarés dans D, ne désignent pas forcément la même chose* que les identificateurs f masqués dans B
- Conséquence :
  - masquage d'une fonction membre de B par une surcharge dans D
  - toutes les surcharges d'une fonction membre doivent être dans une même portée

*Ce ne sont pas forcément des redéfinitions!


```
class B
{public :
   int f(int);
};
```

```
class D : public B
{public :
   int f(int,int);
};
```

```
D d;
d.f(3); // NON!!!!
```

Possibilité de ramener un identificateur (masqué) de B dans la portée de D (utilisation d'une *using-declaration*)

```
class B
{public :
   int f(int);
};
```

```
class D : public B
{public :
   int f(int,int);
   using B::f;
};
```

```
D d;
d.f(3); //OK
```


Mais revenons au polymorphisme…

- Le polymorphisme ne pourra être mis en œuvre qu'à travers des pointeurs ou des références (normal!)
- Pour pouvoir faire du polymorphisme il est nécessaire de connaître le type dynamique d'un pointeur ou d'une référence, c'est à dire le type de effectivement pointé ou référé lors de l'exécution!

Type statique / Type dynamique

- Si on considère un pointeur ou une référence sur un objet de type A,
  - le type statique de l'objet pointé ou référencé est A (type connu à la compilation)
  - mais l'objet pointé ou référencé est-il réellement du type A ou d'un type dérivé?
- La réponse à cette question ne peut être fournie qu'à l'exécution : résolution dynamique du type exact de l'objet
- Le type dynamique d'un objet pointé peut changer au cours du programme


```
class Cadre : public Employe { … };

Cadre c;
Employe e;
Employe & re=c;
Cadre & rc=c;
Employe *pe=&e;
pe=&c;
```

Quels sont les types statiques et dynamiques de c, e, re, rc et pe au cours du programme?

- Le type dynamique d'une instance de la classe A
  - ne change pas au cours de l'exécution du programme
  - coïncide avec le type statique A
- Le type dynamique de l'objet pointé par un A*
  - ne peut être résolu qu'à l'exécution du programme
  - peut varier au cours de son exécution
  - ne coïncide pas forcément avec le type statique A*
- Le type dynamique de l'objet référencé par un A&
  - ne peut être résolu qu'à l'exécution du programme
  - ne coïncide pas forcément avec le type statique A&

- Par défaut, la résolution de l'appel à une fonction membre **accédée à travers un pointeur** A* **ou une référence** A&
  - se fait à la compilation,
  - d'après le type statique du pointeur ou de la référence
- Possibilité de résoudre l'appel à cette fonction membre sur la base du type dynamique de l'objet effectivement pointé :
  - Il suffit de faire précéder la fonction membre de A du qualificatif virtual
  - permet l'aiguillage vers une éventuelle redéfinition de cette fonction membre

119

119

Fonctions virtuelles

```
Class Employe
{public :
  virtual void affiche()
   {std::cout<< num
              << std::endl;}
 private :
   int num;
};
```

```
class Cadre : public Employe
{public :
  virtual void affiche()
    { Employe::affiche();
      std::cout<< echelon
               <<std::endl;
    }
 private :
   int echelon;
};
```

```
Employe e; Cadre d;
Employe & re=d; Employe *pe=&e;
 e.affiche(); d.affiche();
 re.affiche(); pe->affiche();
 e=d; e.affiche();
 pe=&d; pe->affiche();
```

120

120

```
//Employe::affiche puis Cadre::affiche
//Cadre::affiche puis Employe::affiche
//Employe::affiche
//Cadre::affiche
```

121

121

- Une classe avec une fonction membre virtuelle est dite polymorphe
- Une fonction membre virtuelle est appelée une méthode
- Une fonction virtuelle reste virtuelle dans les classes dérivées
  - inutile de répéter le mot clef virtual
- Attention:
  - A la compilation, vérification de la **validité** de l'appel à une méthode, **sur la base du type statique**
  - Une fonction virtuelle peut être masquée par un identificateur identique qui ne correspond pas à une redéfinition …

122

122

```
class Employe
{ …
   virtual void affiche();
   virtual int age_retraite();
   virtual void augmentation(int);
   virtual void salaire();
};
Employe *pe=new Cadre;
pe->affiche();
pe->age_retraite();
pe ->age_retraite(2);
pe ->augmentation(5.5);
pe->salaire();
```

```
class Cadre : public Employe
{ …
   void affiche();
   int age_retraite(int);
   void augmentation(double);
};

Cadre *pc=new Cadre;
pc->affiche();
pc->age_retraite();
pc ->age_retraite(2);
pc ->augmentation(5.5);
pc->salaire();
```

Quelles instructions sont valides?
Quelles sont les fonctions effectivement appelées?

123

123

```
class Employe
{ …
   virtual void affiche();
   virtual int age_retraite();
   virtual void augmentation(int);
   virtual void salaire();
};
Employe *pe=new Cadre;
pe->affiche(); // C::affiche()
pe->age_retraite();//E::age_retraite()
pe ->age_retraite(2);
pe ->augmentation(5.5);//E::aug(int)
pe->salaire(); // E::salaire()
```

```
class Cadre : public Employe
{ …
   void affiche();
   int age_retraite(int);
   void augmentation(double);
};

Cadre *pc=new Cadre;
pc->affiche(); // C::affiche()
pc->age_retraite();// existe …
   // mais masquée à l'analyse statique!
pc ->age_retraite(2);//C::age_ret.(int)
pc ->augmentation(5.5);//C::aug(dble
pc->salaire(); // E::salaire()
```

124

124

- Pour s'assurer qu'on a bien fait une redéfinition et pas juste une surcharge qui vient masquer l'existant…
- C++ 11 est arrivé!

- Redéfinition plus explicite des fonctions membres

```
struct Base {
  virtual void fonc(int);
};
struct Derivee : Base {
   virtual void fonc(int) override;
  // Le compilateur ralera si ce n'est
  // pas une vraie redéfinition ☺
};
```

125

125

Quid des fonctions membres non virtuelles?

```
class Figure
{
  Couleur couleur;
 public :
  virtual void afficher(); //Redéfinie dans classes dérivées
  void effacer();      //Pas virtuelle mais ...
};

void Figure::effacer()
{Couleur temp=couleur;
 couleur=couleurFond;
 afficher();          //appel à une méthode virtuelle
 couleur=temp;
}
Figure *f = new Losange;
f->afficher(); f->effacer();
```

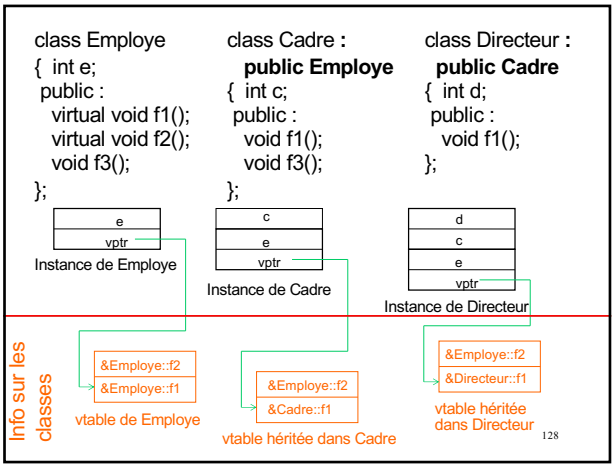
126

Liaison dynamique, comment?

- Le polymorphisme a un coût :
  - Chaque **classe** d'une hiérarchie polymorphe est caractérisée par une table des fonctions virtuelles **de la classe**
  - encombrement supplémentaire des instances des classes polymorphes
    - présence d'un **champ supplémentaire** de type pointeur, permettant d'accéder à la **table virtuelle** de la classe
    - la table virtuelle contient les adresses des fonctions virtuelles de la classe
  - délai supplémentaire à l'appel puisque appel indirect

127

127



128

128

Cadre d'utilisation

- Quand définir une fonction membre comme virtuelle ?
  - si **redéfinition** dans des classes dérivées
  - si accès aux objets des classes dérivées via des **pointeurs** ou des **références**
- Autrement, le comportement d'une fonction virtuelle n'est pas différent de celui d'une fonction ordinaire (mais son appel est plus lent!)

129

129

Pour empêcher la dérivation d'une classe

- Depuis C++11, mot clef final

```
struct Base final {
   blablacode
 };

 //struct Derivee : Base { };
```

Dérivation refusée par le compilateur

final existait déjà en JAVA pour désigner une méthode non redéfinissable ou une classe non dérivable

130

130

- Polymorphisme mais analyse statique à la compilation
  - vérification statique de la **validité des droits**
  - prise en compte statique des **arguments par défaut**
  - résolution statique des **surcharges éventuelles**

→ **Le type statique détermine la signature de la fonction appelée**

131

131

```
class Employe
{
public :
   virtual void f1();
   virtual void f2(int i=0);
   virtual void f3(int);
};
```

```
class Cadre : public Employe
{private :
   void f1();
 public :
   void f2(int);
   void f3(int);
   void f3(char);
};
```

```
Employe *e=new Cadre;
e->f1();
e->f2();
e->f3('a');
```

```
Cadre *c=new Cadre;
c->f1();
c->f2();
c->f3('a');
```

Quelles instructions sont valides?
Quelles versions des fonctions sont appelées?

132

132

```
class Employe
{
public :
   virtual void f1();
   virtual void f2(int i=0);
   virtual void f3(int);
};
```

```
class Cadre : public Employe
{private :
   void f1();
 public :
   void f2(int);
   void f3(int);
   void f3(char);
};
```

```
Employe *e=new Cadre;
e->f1(); //Cadre::f1()
e->f2(); //Cadre::f2(0)
e->f3('a'); //Cadre::f3((int)'a')
```

```
Cadre *c=new Cadre;
c->f1();// private!
c->f2();
c->f3('a'); // Cadre::f3('a')
```

133

133