

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

Norme ISO

Raphaëlle Chaîne
raphaelle.chaine@liris.cnrs.fr
2023-2024

1

1

Éléments de contrôle pour les classes

- Amitié (*friend*)
- Qualificatif *explicit*

91

91

Fonction amie d'une classe

- Fonction autorisée à accéder à tous les membres des instances d'une classe dont elle est l'amie
- Une fonction amie est déclarée comme telle dans la spécification de la classe
 - syntaxe :
`friend` + prototype fonction amie;
- Une fonction peut être amie de plusieurs classes

92

92

```
class Complexe
{ //declaration amie
  friend bool identique_re(Complexe,Complexe);
  ...
};
bool identique_re(Complexe z1,Complexe z2)
{ return z1.re==z2.re; // acces membres }

puis ...
Complexe z(3)
if (identique_re(z,Complexe(3,5)))
{ ... }
```

93

93

Classe amie

- Classe dont toutes les fonctions membres sont amies
- Une classe amie est déclarée comme telle dans la définition de la classe

```
class CC
{
  friend class DD;
  ...
};
```

94

94

explicit

95

95

Mot-clef *explicit*

- Possibilité de conversion implicite induite par un constructeur à un seul argument : parfois indésirable !
- La supprimer avec le qualificatif *explicit* (lors de la déclaration du constructeur)

96

96

- Exemple
class Tableau
{public :
 explicit Tableau(int t) : tab(new int[t]), taille(t)
 { }
 ...
private :
 int * tab;
 int taille;
};

- Suppression de la possibilité (absurde) de conversion implicite d'un int en Tableau

97

97

Objets fonctions

98

98

Surcharge de l'opérateur ()

- Les objets instances d'une classe surchargeant l'opérateur () sont appelées des "objets fonctions"
- Elles peuvent être utilisées comme des fonctions
- On parle aussi de foncteur : abstraction de la notion de fonction

99

99

```
class Convertisseur  
{public :  
    Convertisseur(double d=1.0) : taux(d) {}  
    double operator () (double sum) const  
    {return sum*taux;}  
private :  
    double taux;  
};  
  
puis ...  
Convertisseur euro2francs(6.56);  
std::cout << euro2francs(1000) << std::endl;  
std::cout << Convertisseur(2.0)(1000) << std::endl;  
std::cout << Convertisseur()(1000) << std::endl;
```

100

100

- Avantages :
 - La fonctionnalité d'un objet fonction peut être paramétrée grâce à ses données membres
 - Les objets fonctions peuvent être transmis en argument d'une autre fonction (sémantique plus précise qu'un pointeur de fonction)
 - Sémantique plus précise d'une fonction qui prend en paramètre un objet fonction de type Convertisseur par rapport à une fonction qui prend en paramètre un pointeur sur une fonction (agissant sur un paramètre de type double et qui retourne un double).

101

101

Redéfinitions de fonctions membres et polymorphisme

102

102

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

103

103

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();
```

104

104

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!

105

105

Redéfinition de fonctions membres

```
class Employe      class Cadre : public Employe
{ ...             { ...
  int f(double);   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 simplifiéeil convient de redéfinir la fonction membre f correspondante
- Déclaration de la version enrichie ou simplifiée de f dans la définition de Cadre

106

106

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 &)Alors il est possible que Cadre::f retourne un TD* (resp TD &) si 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
```

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)
```

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

110

110

Rassurez vous le polymorphisme existe tout de même en C++...

- Le polymorphisme pourra être mis en œuvre mais uniquement 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!

111

111

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

112

112

```
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?

113

113

- 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*

114

114

- 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

115

115

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();

```

116

116

```

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

```

117

117

- Une classe avec une fonction membre virtuelle est dite **polymorphe**
- Une fonction membre virtuelle est appelée une **méthode**

118

118

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

119

119

```

class B      class D : public B      D d;
{public :   {public :                          d.f(3);
  int f(int);   int f(int); //Redefinition
};           };

```

Et là?

```

class B      class D : public B      D d;
{public :   {public :                          d.f(3); ??
  int f(int);   int f(int,int); //Surcharge
};           };

```

120

120

- **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!

121

121

```

class B      class D : public B      D d;
{public :   {public :                          d.f(3); // NON!!!!
  int f(int);   int f(int,int);
};           };

```

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

```

class B      class D : public B      D d;
{public :   {public :                          d.f(3); //OK
  int f(int);   int f(int,int);
};           using B::f;
};           };

```

122

122

Mais revenons au polymorphisme...

- **Attention:**
 - A la compilation, choix de la **signature de la fonction membre appelée** et de la **validité de son appel** sur la base du **type statique du pointeur ou de la référence!!!!**
 - **MEME QUAND IL S'AGIT D'UNE FONCTION MEMBRE VIRTUELLE!** (virtual)

123

123

```

class Employe      class Cadre : public 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();
Quelles instructions sont valides?
Quelles sont les fonctions effectivement appelées?

```

124

124

```

class Employe      class Cadre : public 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); // E::age_retraite(2)
pe->augmentation(5.5); // E::aug(int)
pe->salaire(); // E::salaire()

```

125

125

- **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 ...

126

126

- 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 ramera si ce n'est
    // pas une vraie redéfinition ☺
};
```

127

127

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();
```

128