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
2023-2024

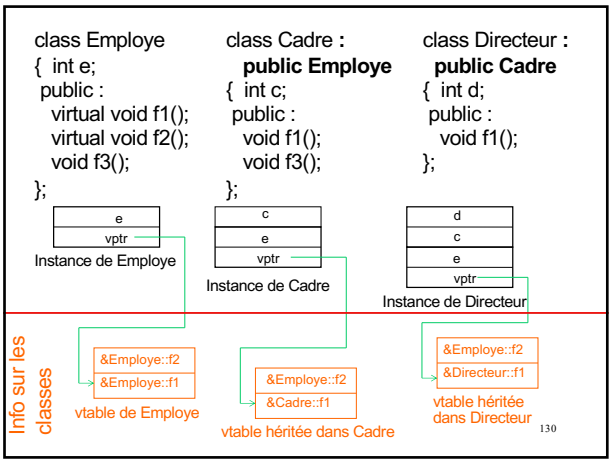
Quid des fonctions membres non virtuelles?

```
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();
```


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



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!)

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

- 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**


```
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?


```
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')
```


Constructeurs et classes polymorphes

- Un constructeur ne peut être virtuel
- Pouvez-vous deviner pourquoi?

```
Employe & gerard = ….
Employe * pe= new Employe(gerard);
Employe e(gerard);
```


Constructeurs et classes polymorphes

- Pour interdire tout appel virtuel dans le constructeur :
- Lors de la création d'une instance d'une classe polymorphe :
  - initialisation du pointeur vers la table des fonctions virtuelles ….
  - **seulement après** la création et l'initialisation de cet objet !
- Dans le corps d'un constructeur, **résolution statique** des appels à une fonction virtuelle


```
struct Employe
{ const char type[50];
 public :
   Employe();
   virtual void specifique()
      { std::strcpy(type,"Employe"); }
   void affiche()
      {std::cout<<type;}
};
```

```
Employe::Employe()
 {//Initialisation commune
  // a tous les Employe :
       … ;
  //Initialisation specifique a
  //chaque type d'Employe :
    specifique(); //this-> specifique();
 }
```

```
struct Cadre : public Employe
{public :
   Cadre(): Employe() {}
   void specifique()
      { std::strcpy(type,"Cadre"); }
};
```

```
Employe e; Cadre c;
Employe *ade=new Cadre;
Cadre *adc=new Cadre;

e.affiche();
c.affiche();
ade->affiche();
adc->affiche();
```

Trace du programme :

Employe
Employe
Employe
Employe

Pourtant …

- Il peut être utile de créer puis d'initialiser un objet, par copie d'un autre objet dont le type n'est pas connu à la compilation
- On ne peut pas compter sur le constructeur par copie

139

139

Essayons tout de même :

```
class Employe
{ … };

class Cadre : public Employe
{ … };

Employe *ade1 = new Cadre;
Employe *ade2 = new Employe(*ade1);
```

- *ade1 est upcasté en employé
- Le type dynamique de l'objet pointé par ade2 est Employe !

140

140

- Création d'objet par copie d'un autre objet de type imprécis à la compilation

```
class Employe
{...
 Employe(const Employe&);
 virtual Employe *clone()
   {return new Employe(*this);}
};

class Cadre : public Employe
{…
  Cadre(const Cadre&);
  Cadre *clone() //redefinition
    {return new Cadre(*this);}
};

Employe *ade1 = new Cadre;
Employe *ade2 = ade1->clone();
// Cadre = type dynamique de *ade2
```

141

141

De même …

- Création d'un objet de même type que celui (inconnu à la compilation) d'un autre objet

```
class Employe
{...
 Employe();
 virtual Employe *nouveau()
   {return new Employe;}
};

class Cadre : public Employe
{…
  Cadre();
  Cadre *nouveau() //redefinition
    {return new Cadre;}
};

Employe *ade1=new Cadre;
Employe *ade2=ade1->nouveau();
// Cadre = type dynamique de *ade2
```

142

142

Destructeurs et classes polymorphes

- **Le destructeur d'une classe polymorphe doit être virtuel** (ex : destructeur de la racine d'une hiérarchie de classe)

```
class Employe
{ int num;
 public:
   Employe(int i) : num(i){}
   virtual ~Employe()
     {std::cout<< "Emp. detruit";}
};

class Cadre : public Employe
{  char * grade;
  public :
    Cadre(int i, char *g) : Employe(i)
      {grade=new char[strlen(g)+1];
       strcpy(grade,g);
      }
    …
    ~Cadre()
      {delete [] grade;
        std::cout<<"Cadre detruit, ";}
};

Employe *e=new Cadre(555,"chef rayon lessive");
delete e;
```

143

143

// A l'affichage : Cadre detruit, **Emp. detruit**

144

144

Destructeurs et classes

- **Le destructeur d'une classe peut être virtuel**
- C'est même obligatoire si la classe est amenée à être dérivée et les instances pointées de manière polymorphe
- En revanche, si une classe n'est pas destinée à être polymorphe (ce qui est notamment le cas des classes qui ne seront jamais dérivées),
  - on peut économiser le coût du polymorphisme
  - avec un destructeur non virtuel
- Et C++11 permet de le garantir de façon explicite

```
struct Base final { blabla };
```

  - Le compilateur refusera ensuite toute dérivation

145

145

Fonction virtuelle pure

- Fonction virtuelle déclarée **mais non définie** au niveau général d'une hiérarchie de classes

```
class Figure
{ ...
  virtual void dessiner() = 0;
};
```

  La fonction dessiner ne sera définie que dans des spécialisations de la classe Figure

- La classe Figure est dite abstraite
- Mot clé abstract en JAVA

146

146

- Une fonction virtuelle pure
  - demeure virtuelle pure au fil des dérivations,
  - aussi longtemps qu'elle ne fait pas l'objet d'une définition
- Obligation de définir une implantation dans une classe dérivée directe ou indirecte (sinon elle est à son tour abstraite)
- Impossibilité de créer des instances d'une classe abstraite

147

147

```
class Figure
   {…
   virtual void dessiner=0;
   };
class Losange : public Figure
   {…
     void dessiner();
   };
```

Quelles définitions sont correctes?

```
Figure f;
Figure *pf;
pf=new Figure;
pf=new Losange;
Losange l;
```

148

148

```
class Figure
   {…
   virtual void dessiner=0;
   };
class Losange : public Figure
   {…
     void dessiner();
   };
```

Quelles définitions sont correctes?

~~Figure f;~~
Figure *pf;
~~pf=new Figure;~~
pf=new Losange;
Losange l;

149

149

En JAVA

- Le masquage (*hiding*) en Java se fait sur la base du nom mais aussi des paramètres d'une méthode (mais pas le type retourné)
  - Plus intelligent que le simple masquage par le nom comme en C++
  - Pas d'effet de bord sur l'installation de surcharges (*overloading*) dans une classe dérivée
- Pour prévenir tout problème en C++:
  - Indiquez vos redéfinitions de fonctions membres avec le mot clé `override` (C++11)
  - toujours accompagner l'introduction d'une surcharge dans une classe dérivée d'une `using` declaration

150

150

Rappel de l'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
```

151

151

En JAVA

- Les annotations Java permettent de signaler l'intention d'une redéfinition (*overriding*) à la manière du *override* de C++11

```
class Cadre extends Employe
{ …
   @Override
  public void affiche()
   { super.affiche(); //affiche de la classe Employe
     system.out.println(" Cadre ");
   }
}
Employe gege = new Cadre;
gege.affiche();
```

152

152

En JAVA

- Attention la liaison dynamique (*late binding*) ne se fait que sur les méthodes d'instances, pas sur les méthodes de classe

153

153

En JAVA

- Exemple légèrement modifié de la doc oracle :

```
public class Animal {
   public static void testClassMethod() {
      System.out.println("The static method in Animal");
   }
   public void testInstanceMethod() {
      System.out.println("The instance method in Animal");
   }
}
```

154

154

En JAVA

- Exemple légèrement modifié de la doc oracle :

```
public class Cat extends Animal {
   public static void testClassMethod() {
      System.out.println("The static method in Cat");
   }
   public void testInstanceMethod() {
      System.out.println("The instance method in Cat");
   }
   public static void main(String[] args) {
      Cat myCat = new Cat();
      Animal myAnimal = myCat;
      myAnimal.testClassMethod();
      myAnimal.testInstanceMethod();
   }
}
```

155

155

En JAVA

- A l'exécution :

```
The static method in Animal
The instance method in Cat
```

Le compilateur opte pour la méthode static correspondant au type statique de la référence myAnimal

156

156

En JAVA

- Utilisation du mot clé **abstract**

```
abstract class Figure
{    …
     public abstract void dessiner();
     public void effacer() { … blablacode …}
}

class Losange extends Figure
{ …
  public void dessiner() {… bliblicode …}
}
```

157

157

- Intérêt des classes abstraites C++
  - Définition d'une interface commune à travers laquelle accéder aux fonctionnalités des sous-classes : classe d'interface
  - Mais aussi possibilité de factoriser des algos incomplets mais communs aux classes dérivées :
    Modèle de conception *(Design Pattern)*
    Patron de méthode

158

158

Patron de méthode

- Définition de la trame générale d'un algorithme au niveau de la classe de base
- Les détails de la trame sont complétés de manière spécifiques dans les classes dérivées

159

159

- Exemple inspiré de la hiérarchie de classe Magnitude en Smalltalk
- Idée : éviter de définir dans toutes les classes de nombreux opérateurs qui peuvent s'obtenir par combinaison les uns des autres

Classe abstraite générique des objets comparables entre eux

```
class Magnitude {
  public :
   virtual  bool operator <(const Magnitude & mag) const =0;
   virtual  bool operator ==(const Magnitude & mag) const=0;
   Patrons de méthodes :
   bool operator >=(const Magnitude & mag) const
     { return !(*this < mag);}
    ……
  };
```

160

Classe concrète :

```
class Entier : public Magnitude {
   private :
     int val;
   public :
    virtual  bool operator <(const Magnitude & mag) const
      { return val < (( const Entier &) mag).val; }
    virtual  bool operator ==(const Magnitude & mag) const
     { return val == (( const Entier &) mag).val; }
   …
   //Pas de redéfinition de operator >=
   …
};
```

161

161

En JAVA

- En Java on peut choisir de travailler avec des classes abstraites ou des interfaces, sachant qu'il est possible de proposer un code par défaut même dans une interface… (mot clé default)
- Attention les interfaces JAVA ne peuvent pas contenir des attributs!
- A vous de bien savoir différencier l'usage des classes abstraites et interfaces!
  - Les interfaces définissent un comportement (Comparable, Runnable, …)
  - Les classes abstraites se concentrent plus sur ce qui fonde la nature intrinsèque d'un ensemble de classe

162

162

En JAVA

Exemple extrait de la doc Oracle :

```
public class Horse {
    public String identifyMyself() { return "I am a horse. ";}
}
public interface Flyer {
    default public String identifyMyself() { return "I am able to fly.";}
}
public interface Mythical {
    default public String identifyMyself() { return "I am mythical.";}
}
public class Pegasus extends Horse implements Flyer, Mythical {
    public static void main(String... args) {
        Pegasus myApp = new Pegasus();
        System.out.println(myApp.identifyMyself());
    }
}
```


163

En JAVA

A l'exécution :

I am a horse

A l'exécution c'est bien entendu l'héritage qui l'emporte!

164

Identification dynamique du type (RTTI*)

- Encombrement mémoire supplémentaire des classes polymorphes :
  - A l'exécution, il est possible de connaître le type dynamique d'un objet pointé
- Possibilité de gestion des types dynamiques
- 2 opérateurs :
  - typeid
  - dynamic_cast<>

(*RTTI = Run Time Type Information)

165

- opérateur **typeid**
  - `#include<typeinfo>`
  - Syntaxe :
    - typeid(*type*)
    - typeid(*expression*)
  - Renvoie une valeur de type **const type_info &**

```
class type_info
   {
      public :
        const char * name() const;
        int operator == (const type_info &) const;
        int operator != (const type_info &) const;
        int before (const type_info &) const;
   };
```

Sans rapport avec héritage

```
Cadre c; Employe *ademp = &c;
std::cout << typeid(c).name()
          << typeid(*ademp).name();
```

166

```
#include <iostream>
#include <typeinfo>

class Employe
{
public :
  virtual ~Employe(){}
};
int main()
{
  const Employe * const  pe = new Cadre();
  Cadre * const  pc = new Cadre();
  std::cout << typeid( pe ).name() << std::endl;
  std::cout << typeid( pc ).name() << std::endl;
  std::cout << typeid( *pe ).name() << std::endl;
  std::cout << typeid( *pc ).name() << std::endl;
  std::cout << (typeid( *pc )== typeid( *pe )) <<
  std::endl;
  return 0;
}
```

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

167

- Solution :

  PC7Employe
  P5Cadre
  5Cadre
  5Cadre
  1

- Pour obtenir des informations sur le type dynamique de l'objet pointé :
  - déréférencer les pointeurs
  - type statique du pointeur sinon!
- Le caractère const (ou non const) du type dynamique n'est pas pris en compte par la classe type_info

168

- Utilisation de typeid sur des classes non polymorphes :
  - Attention : Erreurs compilations ou informations sur le type statique
    (… sauf existence de “enable RTTI”)

169

169

- Un opérateur fiable de *downcast* (ou d’*upcast*) : **dynamic_cast**<*type*>(*expression*)
  - Conversions de pointeurs au sein d’une hiérarchie de classes polymorphes
  - **Tentative de conversion d’un Employe* en Cadre* (spécialisation,** *downcast***)**

```
Employe * ademp = …;
Cadre* adcad=dynamic_cast<Cadre*>(ademp);
```

    - rend l’adresse de l’objet Cadre effectivement pointé par ademp **si c’est possible**,
    - 0 sinon
  - **Conversion d’un Cadre* en Employe* (généralisation,** *upcast* **)**
    standard

(où Cadre dérive publiquement de Employe)

170

- S’applique aussi aux références
  - **Tentative de conversion d’un Employe& en Cadre& (spécialisation)**

```
Employe & b= …;
Cadre &d=dynamic_cast<Cadre &>(b);
```

    - Établit la référence sur objet de type Cadre **si c’est possible**,
    - levée d’une exception **std::bad_cast** sinon (nécessite #include<typeinfo>)
  - **Conversion d’un Cadre& en Employe& (généralisation)**
    standard

171

171