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

Constructeurs et classes polymorphes

- Un constructeur ne peut être virtuel
- 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


```
class 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();
 }
```

```
class 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


```
class Employe
{ … };

class Cadre : public Employe
{ … };

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

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

- 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
```

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
```

139

139

Destructeurs et classes polymorphes

- **Le destructeur d'une classe peut (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;
```

140

140

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

141

141

Destructeurs et classes polymorphes

- **Le destructeur d'une classe peut être virtuel**
- C'est même recommandé si la classe est amenée à être dérivée….
- En revanche, si une classe n'est pas destinée à être dérivée,
  - 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

142

142

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

143

143

- 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

144

144

```
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;
```

145

145

```
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;
```

146

146

En JAVA

- Le masquage (*hiding*) en Java se fait sur la base du nom mais aussi des paramètres d'une méthode
  - Plus intelligent que le simple masquage par le nom comme en C++
  - Permet d'installer des surcharges (*overloading*) dans une classe dérivée sans avoir à utiliser une *using* déclaration pour ne pas masquer les méthodes de la classe de base
- Conclusion :
  - En C++ toujours accompagner l'introduction d'une surcharge dans une classe dérivée d'une using declaration

147

147

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
```

148

148

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();
```

149

149

En JAVA

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

150

150

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");
    }
}
```

151

151

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();
    }
}
```

152

152

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

153

153

En JAVA

- Utilisation du mot clé **abstract**

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

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

154

154

- 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

155

155

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

156

156

- 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);}
    ……
  };
```

157

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 >=
   …
};
```

158

158

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

159

159

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());
      }
}
```

160

160

En JAVA

A l'exécution :

I am a horse

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

161

161

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)

162

- 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();
```

163

```
#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
{};
```

164

- 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

165

165

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

166

166

- 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)

167

- 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

168

168

– Bilan sur les 4 opérateurs de transtypage (cast)
  • Pour remplacer les casts “à la C”
  • A chaque opérateur, une sémantique

– 2 opérations à ne pas utiliser dans un logiciel de qualité industrielle :
  • **reinterpret_cast :**
    Conversion non standard et non sûre de valeurs
    (entre pointeurs sans rapport et/ou entre pointeurs et entiers)
    reinterpret_cast<long>(&a);
  • **const_cast :**
    Pour supprimer, à la compilation, le caractère **const** d’une expression (à vos risques et périls!)
    const Titi &t=tt; f(const_cast<Titi &>(t));
    A ne faire que si on sait que f ne modifie pas t ….

169

169

– Opérations de transtypage pouvant être indispensables :

  • **static_cast :**
    Conversion plausible de valeurs (conversions numériques, *downcast* statiques dont on est sûrs*)* avec analyse statique à la compilation
    int i; double d=static_cast<double>(i);
    Employe &re=theboss; …
    puis *static_cast<Cadre &>(re)* quelque part dans le code

  • **dynamic-cast () :**
    downcast avec vérification statique puis dynamique à l’exécution (moyennant RTTI)

```
Employe &re=theboss; …….
try{  ……..
      ….  dynamic_cast<Cadre &>(re);
    }
catch(std::bad_cast)
     {  ……. }
```

170

170

- Quel opérateur choisir pour un *downcast* ?
  – *static_cast* plus efficace mais moins sûr que *dynamic_cast*
  – Certains appels à *dynamic_cast* ne peuvent être remplacés par un *static_cast*
  – Idée :
    • Remplacer tous les *downcast* qui le supportent par une macro qui utilise **dynamic_cast** en mode *debug* et **static_cast** en mode *release*

171

171

```
#ifndef DEBUG
#define CAST_PTR(Type , toType , expr ) \
    ( static_cast < toType * > ( expr ) )
#else
#define CAST_PTR(Type , toType , expr )  \
   ( ( (dynamic_cast< toType *>(expr)) !=0 ) ?     \
       dynamic_cast< toType *>(expr) :  \
       erreurconversion(#expr,  __FILE__,  __LINE__ ), \
       static_cast< toType *>(0)
    )
#endif
```

Où *erreurconversion()* est une fonction qui livre des indications sur la localisation de l’erreur avant d’interrompre le programme.

CAST_PTR(Employe,Cadre,pe);

172

172

Les limites du polymorphisme en programmation objet …

- Le polymorphisme ne s’applique que sur l’argument implicite d’une fonction membre
  – argument privilégié du traitement concerné…
  – il n’en est pas de même pour les autres arguments!!!
- Concernant certains traitements s’appliquant à plusieurs arguments, il n’y a parfois aucune raison qu’un argument soit privilégié plutôt qu’un autre…
  – Exemple : la plupart des opérations mathématiques (Addition de deux nombres, etc.)

173

173

Difficulté de créer des opérateurs binaires polymorphes

```
class Produit
{ int prix;  …
};
class ProduitFrais : public Produit
{ date peremption; …
};
Produit *adp1= … ;
Produit *adp2= … ;
*adp1=*adp2;
```

ie. adp1->operator=(*adp2);

Comment faire pour que l’opération d’affectation s’adapte au type dynamique de ses 2 opérandes?

174

174

- **Adaptation à l'opérande de gauche** :
  - Surcharger l'opérateur = comme opérateur virtuel de la classe Produit

```
class Produit
{
  int prix;
 public :
   virtual Produit & operator = ( const Produit & );
};
```

  - … redéfini dans la classe ProduitFrais

```
class ProduitFrais : public produit
{
  date peremption;
 public :
   ProduitFrais & operator = ( const Produit & );
};
```


- **Adaptation à l'opérande de droite** :
  1ère solution

  - Pour chaque spécialisation de l'opérateur d'affectation, **appel à une fonction membre virtuelle** invoquée sur l'opérande de droite :

```
Produit & Produit::operator = (const Produit &  opdroit )
  {
    prix=opdroit.prix; …;
    return *this;
  }
ProduitFrais & ProduitFrais::operator = (const Produit & opdroit )
  {
    opdroit.estAffecteA(*this);
    return *this;
  }
```

où *estAffecteA(ProduitFrais &) const* est une fonction membre virtuelle de la classe Produit

- Désavantages :
  - Dans la définition de la classe de base, nécessité d'en connaître toutes les spécialisations ultérieures …

```
class Produit
{ …
 virtual Produit & operator = (const Produit & );
 virtual void estAffecteA(ProduitFrais &) const;
 virtual void estAffecteA(ProduitFragile &) const;
  …
};
```

  - Nécessité de retoucher à la classe de base à chaque nouvelle dérivation !

- **Adaptation à l'opérande de droite** :
  2ème solution

  -Utilisation de l'opérateur dynamic_cast<>

```
Produit & Produit::operator =(const Produit & p)
 { prix=p.prix;
   …
   return *this;
 }

ProduitFrais & ProduitFrais::operator =(const Produit & p)
 {
   ProduitFrais *temp;
   if ((temp=dynamic_cast< const ProduitFrais *>(&p))!=0)
      { … }
   else
     { …// retour d'erreur?}
    return *this;
 }
```