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

1

Buts de l'UE

- Introduire généricité et méta-programmation dans le développement logiciel de librairies
- Comment et à quel coût?
- Dans un langage où il est possible d'activer ou non les fonctionnalités dont on a besoin
  - Intérêt de C++
  - Mise en place de bibliothèques comme la STL
  - Différences fondamentales avec JAVA

2

Langages à maîtriser

- Rapport annuel Institute of Electrical and Electronics Engineers (IEEE)
  - Python, C++, C et Java en 2018,
  - Python, Java, C et C++ en 2019,
  - … ces 4 langages se tirent la bourre depuis des années mais Python 1er depuis 3 ans (Deep learning : Keras, TensorFlow, CNTK et Theano…)
- Excellence de la France pour la maîtrise du C++
  - http://www.lemondeinformatique.fr/actualites/lire-la-france-compte-les-meilleurs-developpeurs-en-c-au-monde-65877.html

3

Programmation C++

- Paradigme de Programmation Orientée Objet (POO), mais pas seulement!
  - Développement de composants logiciels réutilisables pour faciliter l'activité de programmation
  - Utilise un certain nombre de notions fondamentales de base :
    Objets, classes, héritage, liaison dynamique ou polymorphisme ...

4

1ers Langages dédiés à la POO

- Smalltalk (1980)
- Simula (1967)
- Eiffel

Langages vraiment Orientés Objet

5

Démarche légèrement différente pour C++

- Conception par B. Stroustrup (AT&T-1982)
- Modification et enrichissement du langage C-ANSI pour permettre l'application de concepts de POO
- Sur-ensemble du langage C-ANSI … avec quelques tolérances en moins

6

Support de différents paradigmes de programmation

- Liberté du programmeur d'adopter un style plus ou moins orienté objet
- Contributions à différents paradigmes de programmation :
  - procédurale,
  - modulaire,
  - générique,
  - Le paradigme de programmation fonctionnelle n'est pas en reste, avec l'apparition des lambda fonctions depuis le standard de 2011

7

7

Support de différents paradigmes de programmation

- Liberté du programmeur d'adopter un style plus ou moins orienté objet
- Différences avec Java (1995)
  - C++ est un langage plus proche de la programmation système
    - utilisation massive de la pile, pointeurs, résolution statique maximale, personnalisation possible de l'allocation dynamique
    - absence de ramasse-miette, classes *template*
  - Bibliothèque standard
    - pas d'interactions prévues avec le monde multimedia ...
  - Compacité et efficacité

8

8

- Forte présence de C++ dans le monde industriel
- Evolution du C++ depuis sa conception jusqu'à sa normalisation
  - Plusieurs publications de l'AT&T(1986, 1987, 1989 et 1991)
  - Extension et enrichissement par le comité ISO : composants standards : fonctions et classes génériques prédéfinies
  - Norme ISO (1998) (maj2003), (2011 puis 2014 et 2017) récemment intégrées et on est en route pour 2020!

9

9

Premiers éléments de différentiation entre C++ et JAVA

- Définition et initialisation d'une instance d'une classe
  - En JAVA

    LaClasse lc = new LaClasse;

    lc est une référence à un objet alloué dynamiquement dans le tas et initialisé par le constructeur de LaClasse

    La désallocation est assurée par un ramasse miette
  - En C++

    LaClasse lc;

    lc est alloué sur la pile et initialisée par appel automatique au constructeur par défaut de LaClasse. L'espace alloué sur la pile est libéré à la fin du bloc de définition de lc, après appel automatique du destructeur

10

10

Premiers éléments de différentiation entre C++ et JAVA

- Définition et initialisation d'une instance d'une classe
  - En C++

    LaClasse* plc = new LaClasse;

    plc est un pointeur créé sur la pile dans lequel on range l'adresse d'un LaClasse alloué dynamiquement dans le tas et initialisé par appel automatique au constructeur par défaut de LaClasse

    delete plc;

    L'espace alloué dans le tas **doit être libéré** par appel explicite à delete. Le destructeur de LaClasse est alors appelé automatiquement sur *plc avant restitution de l'espace mémoire

11

11

Premiers éléments de différentiation entre C++ et JAVA

- Dans les classes
  - En Java

    this est une référence sur l'instance courante

    Utilisation :

    this.attribut1

    this.methode1()
  - En C++

    this est un pointeur sur l'instance courante

    Utilisation :

    *this représente l'instance courante

    this->donnee_membre1

    this->fonction_membre1()

12

12

Premiers éléments de différentiation entre C++ et JAVA

- Les références
  - En JAVA **tous les objets alloués** sont manipulés via des références (pas de symbole dédié)
  - En C++ : utilisation du symbole &

```
int a=5;
int &b=a; // b est un alias de a
b=10; // équivalent à a=10;
```

13

13

Premiers éléments de différentiation entre C++ et JAVA

- Les références
  - En JAVA, les références peuvent changer de cible au cours de l'exécution
  - En C++, les références ne peuvent changer de cible au cours de l'exécution

```
int a=5;
int c=3;
int &b=a;
  // b est un alias de a
  // et ne pourra pas devenir un alias de c
```

14

14

Premiers éléments de différentiation entre C++ et JAVA

- Les références
  - En C++

```
int a = 5;
int &b=a; // b est un alias de a
```

  - La mise en œuvre des références se fait via l'utilisation d'un pointeur masqué, dont b correspond au déférencement
  - A priori, la création de références ne peut se faire que sur une *lvalue*

    (expression que l'on peut mettre à gauche (*left*) de l'opérateur d'affectation)

lvalue = rvalue

15

15

Introduction à la notion de value-ness en C++

- *lvalue* (*l* comme *left*)
  - (Référence sur) une variable (dans la pile) ou un espace alloué dynamiquement (dans le tas) (ie. un « contenant ») **doté d'une adresse**
  - Tout ce qui pouvait traditionnellement être à gauche de l'opérateur d'affectation
- *rvalue* (*r* comme *right*) (*prvalue* depuis C++11)
  - Les valeurs (ie. contenus) que l'on peut mettre dans une *lvalue* (ex : 5)
  - Ces valeurs peuvent très bien être retournées par une fonction
  - Une valeur n'a pas d'adresse

16

16

Introduction à la notion de value-ness en C++

- Introduction à la notion de *value-ness* en C++
  - Quelle est dans ce cas la *value-ness* des « contenants temporaires » créés sur la pile pour retourner la valeur d'une fonction?
  - *xvalue* (expiring value) (depuis C++11)
    - pour désigner la *value-ness* de ces contenants qui n'ont pas de nom et **qui vont être détruits à la fin de l'instruction**
    - **pas grave si on en modifie un peu le contenu!**
  - La fin de l'histoire plus loin dans l'UE ☺

17

17

Introduction à la notion de value-ness en C++

- Introduction à la notion de *value-ness* en C++
  - Depuis C++11
    - *lvalue* et *xvalue* forment la catégorie des *glvalues* (generalized left values)
    - *xvalues* et *prvalues* forment la catégorie des *rvalues*

18

18

Premiers éléments de différentiation entre C++ et JAVA

- Retour sur les références
  - En C++, il est également possible de construire des références sur constantes

```
int a = 5;
const int &b=a;
  // b est un alias de a,
  //mais le contenu de a ne peut être modifié
  // via l'identificateur b
a=10; // mais  b=10; impossible
```

19

19

Premiers éléments de différentiation entre C++ et JAVA

- Retour sur les références
  - Références sur constantes
  - Attention : Il est également possible d'écrire :

```
const int & b=6;
```

  - Est-ce que cela vous surprend?

20

20

Premiers éléments de différentiation entre C++ et JAVA

- Retour sur les références
  - Références sur constantes
  - Attention : Il est également possible d'écrire :

```
const int & b=6;
```

  - **Création d'un temporaire contenant la valeur 6 et dont la durée de vie sera celle b**
  - Utile pour les passages de paramètres encombrants

```
double min(const double & d1, const double & d2);
Possibilité d'appeler min(10.0, 15.0);
// Impossible en C avec les pointeurs
```

21

21

Autres buts du cours

- Comprendre la puissance des *templates*, des *namespaces* et des fonctions virtuelles
- Dans le développement de bibliothèques logicielles génériques comme la STL, BOOST, …
- Comprendre la différence avec JAVA

22

22

- Mieux développer avec C++ quel que soit le paradigme de programmation adopté
  - Programmation structurée procédurale
    - Typage fort, conversions, références, *template*
  - Programmation modulaire
    - Compilation séparée, *namespace*
    - Un module (.h, .cpp) peut correspondre à plusieurs classes formant un tout cohérent
  - Types Abstraits de Données
    - Unité syntaxique représentée par la classe
  - Programmation orientée objet
    - Hiérarchie de classe, polymorphisme

23

23

- Mieux comprendre certains aspects parfois mal connus mais essentiels du langage C++
  - Mécanismes virtuels (pivot du polymorphisme)
  - RTTI
  - Singularité de l'héritage multiple
  - Héritage virtuel
- Les fonctions membres virtuelles constituent le fondement de la flexibilité du C++ au sens de la POO

24

24

– Généricité dynamique (à l'exécution) : Polymorphisme mis à profit dans le modèle de conception (*Design Pattern*) « Patron de méthode » (*Method template*)
– Généricité statique : Les *template* représentent une autre forme de généricité statique mise en œuvre à la compilation
– Bien distinguer la connaissance statique de la connaissance dynamique que l'on peut avoir d'un objet

25

25

- Programmation générique et notion de concept
- Etude de la STL et des concepts sur lesquels elle repose (conteneur, itérateur, algorithme, stratégie, adaptateur)
- Modèles de conception *(Design Pattern)* les plus utilisés
  – Classification, abstraction des types de problèmes que l'on peut rencontrer
  – Bonne pratique d'arrangement de modules
  – Observateur, fabrique, fabrique abstraite, médiateur, singleton, composite, visiteur …

26

26

Programmation orientée objet

- Programmation objet :
  – programmes organisés comme des ensembles d'objets coopérants ensemble
  Objet :
    - Entité fermée douée de mémoire et de capacité de traitement
    - Agissant sur réception de message
    - Pouvant fournir un résultat

  (ex : objets créés à partir de types abstraits de données)
  – possibilités de hiérarchie entre les types et de polymorphisme

27

27

- En C++
  Objets = variables instanciées à partir de classes (class ou struct)
- Possibilité de créer des classes
  – soit "à partir de rien", par composition d'un ensemble de champs (de types primitifs ou de types définis par l'utilisateur) qui pourront être publics ou private
  – soit par dérivation de classes existantes, dites classes de base

28

28

- Intérêt de la dérivation :
  – Reprendre les caractéristiques des classes de base
    - en ajoutant ou modifiant certaines fonctionnalités
    - sans remettre en question ces classes de bases (Inutile de les recompiler, seul l'accès à leur définition est indispensable)
  – Factoriser dans une classe unique les analogies communes à un ensemble de classes
  (Dérivation comme outil de spécialisation croissante)

29

29

- Remarque :
  – Le principal intérêt des hiérarchies de classes est le polymorphisme,
    - Possibilité de manipuler un objet sans connaître son degré exact de spécialisation
  – mais on verra que la mise en œuvre du polymorphisme n'est pas automatique comme en Java!

30

30

- Syntaxe de l'héritage en Java :

```
class Cadre extends Employe
{
 // blabla code
}

Employe gege = new Cadre;
```

31

31

- Syntaxe héritage en C++

```
class Derivee : typederivation Base1,
                typederivation Base2,
                ...
                typederivation Basen
{
 // définitions des membres attributs
 //  et déclarations des fonctions membres
 // spécifiques à la classe Derivee
};
```

- En général *typederivation* = public
- Héritage simple (1 seule classe de base) … ou multiple

32

32

- Exemple :

```
class Employe
{public :
   void affiche();
 private :
   int num;
};
```

puis

```
class Cadre : public Employe
{public :
   void affiche_echelon();
 private :
   int echelon;
};
```

La classe Cadre hérite de la classe Employe, avec ajout de membres spécifiques à la classe Cadre

33

33

- Attention :
  La définition de la classe de base doit être accessible

```
class Employe; //simple déclaration!
class Cadre : public Employe // NON
{
 blabla …
};
```

Remarques

- Définition de la classe Employe dans un .h
- Pas besoin en revanche de la définition des fonctions membres dans le .h (les mettre dans un .cpp), juste de la déclaration des fonctions membres externes

34

34

- Remarque :
  - Les classes dérivées peuvent à leur tour servir de base à une dérivation *

    * à condition que le graphe d'héritage ne contienne pas de cycle!

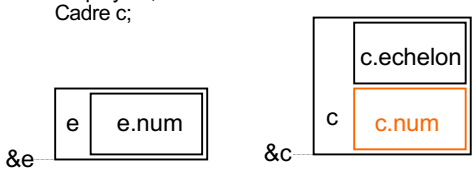
  - Les classes de base
    d'une classe de base
    d'une classe D
    sont aussi des classes de base de D ...
  - En cas d'ambiguïté, parler de **classes de base directes** pour les classes de base héritées explicitement

35

35

- Structure de la classe dérivée
  - Les données membres de la classe de base deviennent des membres de la classe dérivée

```
Employe e;
Cadre c;
```



  - Les fonctions membres de la classe de base deviennent des membres de la classe dérivée

36

36

- *Quand on hérite peut-on toujours toucher au magot?*
- Quel accès à des membres privés hérités?
  - une classe dérivée n'a pas accès aux membres private hérités d'une classe de base
  - accès bloqué **indépendamment du type d'héritage**
- Exemple :

```
void Cadre::affiche_echelon()
{ std::cout << num << std::endl; // NON
                  // num = donnée private de la base
  std::cout<< echelon << std::endl;
}
```

- Conséquence : Les instances des classes dérivées possèdent des membres inaccessibles

- Cette protection est raisonnable, sans quoi il suffirait de dériver une classe, pour accéder à ses données et fonctions membres private
- Rappel :
  - membres private d'une classe accessibles **uniquement** dans le corps des fonctions membres ou amies
  - Une classe est **seule** maître du choix des classes et fonctions autorisées à accéder à ses membres private

- Membres protected
  - Existence d'un niveau intermédiaire d'accès aux membres d'une classe : protected

```
class CC
{ public : Déclarations/définitions interface
  protected :  } Déclarations/définitions partie privée
  private :
};
```

  - Les membres protected d'une classe sont accessibles uniquement :
    - dans le corps des fonctions membres ou amies de la classe
    - dans le corps des fonctions membres ou amies des classes dérivées *

*Indépendamment du type de dérivation, pour les dérivées directes


```
class Employe
 {public :
    void affiche();
  protected :
    int num;
 };

void Cadre::affiche_echelon()
 { std::cout << num << std::endl; //OK
                  // car num protected seulement!
   std::cout<< echelon << std::endl;
 }
```

```
class Cadre : public Employe
   {public :
     void affiche_echelon();
    private :
     int echelon;
   };
```

Mais num reste inaccessible aux utilisateurs des classes Employe et Cadre

Différents types d'héritages

- Il y a 3 types d'héritages :
  - public
  - protected
  - private (dérivation par défaut)
- Quel que soit le type d'héritage, les membres public et protected d'une classe de base sont **accessibles** aux classes dérivées **directement**
- MAIS le type d'héritage choisi peut entraîner une **restriction d'accès** aux membres hérités, **en tant que membres de la classe dérivée** *

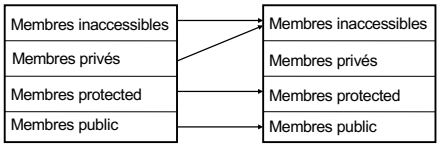
*incidence en cas de nouvelle dérivation

- Cadre usuel d'un héritage public
  - Héritage de l'interface :
    Les membres public de la base demeurent public dans la classe dérivée
  - On peut appliquer à un objet de la classe dérivée toutes les fonctions membres public de la classe de base

```
e.affiche();
c.affiche(); //OK
c.affiche_echelon();
```

Héritage public et évolutions d'accès

- Conservation d'accès pour les membres protected et public
- Les membres private deviennent inaccessibles



Classe de base B — *Classe dérivée D*
*class D : **public** B*

43

43

- Cadre d'un héritage privé (protected ou private)
  - Il existe un type d'héritage dit "privé" protected ou

```
class Employe
{public :
   void affiche();
 private :
   int num;
};
```

```
class Cadre : private Employe
{public :
   void affiche_echelon();
 private :
   int echelon;
};
```

  - Dans ce cas, les membres public de la classe de base deviennent privés dans la classe dérivée

44

44

```
Employe e;
Cadre c;
e.affiche(); // OK
c.affiche(); // NON
c.affiche_echelon(); // OK
```

- Intérêt : L'héritage privé sert à réaliser l'implantation d'une classe dérivée, mais pas à composer son interface
- Exemple d'utilisation
  - Définition d'une classe Pile par dérivation d'une classe Tableau
  - La classe Tableau fournit l'implantation de la Pile, mais pas son comportement

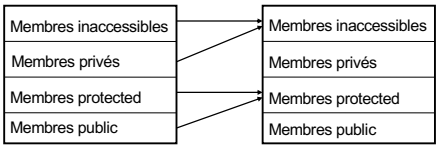
  → choix d'un héritage privé

45

45

Héritage protected et évolutions d'accès

- Conservation d'accès pour les membres protected
- Disparition de l'interface de la classe de base au profit de membres protected



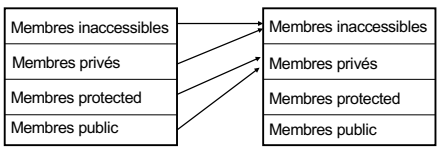
Classe de base B — *Classe dérivée D*
*class D : **protected** B*

46

46

Héritage private et évolutions d'accès

- Disparition de l'interface de la classe de base au profit de membres private
- Les membres protected deviennent private



Classe de base B — *Classe dérivée D*
*class D : **private** B*

47

47

Héritage partiellement public

- Possibilité de rendre son niveau d'accès initial (protected ou public), à un membre hérité non publiquement
- Utilisation d'une using-declaration

```
class Tableau
{ public :
     int taillemax();
  ...
};

Pile p;
p.taillemax(); //OK
```

```
class Pile : private Tableau
{public :
     using Tableau::taillemax;
  ...
};
```

48

48

Que penser de :

```
class B1
{public : void f1(); };

class B2
{public : void f2(); };
```

```
class D : protected B1, B2
{public void g(); };

class DD : public D
{public void gg(); };

void D::g()
     {f1(); f2();}
void DD::gg()
     {f1(); f2(); g()}
```

```
D d; d.f1(); d.f2(); d.g();
DD dd; dd.f1(); dd.f2(); dd.g(); dd.gg();
```

49

49

Réponse :

```
class B1
{public : void f1(); };

class B2
{public : void f2(); };
```

```
class D : protected B1, B2
{public void g(); };

class DD : public D
{public void gg(); };

void D::g()
     {f1(); f2();}
void DD::gg()
     {f1(); f2(); g()}
```

```
D d; d.f1(); d.f2(); d.g();
DD dd; dd.f1(); dd.f2(); dd.g(); dd.gg();
```

f1 : public dans B1, protected dans D, protected dans DD
f2 : public dans B2, private dans D, inaccessible dans DD
g : public dans D et dans DD
gg : public dans DD

50

50

Conversions *upcast* implicites

- Dans le cadre d'un héritage public, possibilité de traiter les instances d'une classe dérivée, comme des instances de la classe de base
- 3 nouvelles conversions standards *upcast* : *

classe dérivée	→	classe de base
pointeur sur instance classe dérivée	→	pointeur sur instance classe de base
référence sur instance classe dérivée	→	référence sur instance classe de base

*Validité dans le contexte d'un héritage public

51

51

```
class Employe
{//définition Employe
};
class Cadre : public Employe
{//définition partie spécifique aux Cadres
};

Employe e; Cadre c;
e=c; // Affectation à e de la partie de c
       // correspondant à un Employe
Employe* ae=&c; // Affectation à ae de l'adresse de la partie
                // de c correspondant à un Employe
Employe & re=c;
```

52

52

- Nuances entre ces différents *upcasts* :
  - convertir un objet de type Cadre en un objet de type Employe revient à lui retirer tous ses membres qui ne font pas partie de Employe (suppression d'information)
  - convertir un Cadre * en Employe *, revient à considérer l'objet pointé comme un Employe, mais il continue à être un Cadre avec tous ses membres (pas de suppression d'information)

53

53

Quelles possibilités d'*upcast* en cas d'héritage non public?

- Les possibilités d'*upcast* peuvent-être vues comme des fonctions membres de la classe dérivée :
  - public,
  - protected
  - ou private

  suivant le mode dont la classe dérivée hérite de sa/ses classes de base

Remarque : En cas de dérivation privée, un *upcast* explicite d'un pointeur est tout de même possible …

54

54

Danger et possibilités explicites de *downcast*

```
class B { … };
class D : public B { … };
B b; D d;
```

- Une instance de B ne peut pas être traitée comme une instance de D
  d=(D)b; // NON (logique!)
- En revanche, étant donné un B*, il est *possible* que l'objet pointé soit en fait un D...
  - Possibilité de downcast d'un B* en D* (ou d'une B& en D&),
  - à n'utiliser que dans les cas où le programmeur sait ce qu'il fait !

55

55

- Downcast d'un B* en D* = opération dangereuse
- Utiliser l'opérateur de transtypage static_cast<D*> de préférence à l'opérateur classique de cast

```
B b; D d;
B* ab=new  D;
D* ad=static_cast<D*>(ab); //OK et a un sens ici
// idem D* ad=(D*)ab;

d=(D)b; //NON Impossible

ab=&b;
ad= static_cast<D*>ab; // OK mais dangereux
                       // car mensonger
d=*(static_cast<D*>)&b; // Vicieux
```

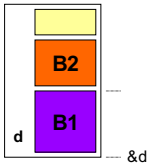
56

56

- En C-ANSI la conversion d'un pointeur
  - de type T1 *
  - en un pointeur de type T2*

  ne modifie pas sa valeur
- En C++, une conversion *upcast* ou *downcast* d'un T1* en T2* peut impliquer la modification de sa valeur

```
class B1{ … };
class B2{ … };
class D : public B1, public B2 { … };
D d;
B1 *p1=&d; B2 *p2 = &d; D* p=&d;
p=static_cast<D*>(p2);
```



57

57

upcast comme outil de généralisation des pointeurs

- Utile pour traitements génériques :
- Ex : Ensemble de classes permettant de gérer un jeu vidéo :

```
class sprite
{ public :
    void printname();
};
class pacman : public sprite
{ … };
class ghost : public sprite
{ … };
```

58

58

- Structures de données pour mémoriser les différents types de sprites présents dans le jeu :

```
pacman * mypac[maxpac];
ghost * myghost[maxghost];
```

- Structure pour mémoriser tous les sprites

```
sprite * mysprite[maxsprite];
```

59

59

- Initialisation de tous les sprite

```
int ipac=0,ighost=0,isprite=0;

mysprite[isprite++]
      = mypac[ipac++]
            = new pacman("pac");
etc.

mysprite[isprite++]
      = myghost[ighost++]
            = new ghost("ghost");
etc.
```

60

60

- Application du traitement générique printname() à tous les sprite

```
for( int i=0; i<isprite ; i++)
{
  mysprite[isprite++]->printname();
}
```

61

61