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

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)

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

Sans rapport avec héritage


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

class Employe
{
public :
  virtual ~Employe(){}
};
class Cadre : public 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;
}
```


- 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

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

- 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

- 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 ….

172

172

- 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)
   { …….. }
```

173

173

- 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*

174

174

```
#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);
```

175

175

Les limites du polymorphisme en programmation objet …

- Le polymorphisme ne s'applique que sur l'argument implicite `this` 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 ne devraient pas être codées en opérateur membre (Addition de deux nombres, etc.)
  - Le cas de l'affectation est différent car le statut de ses 2 arguments est différent.

176

176

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?

177

177

- **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 & );
};
```

178

178

- **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

179

179

- 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 !

180

180

- **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
     { …// comportement different de l'affectation à un produit frais}
   return *this;
 }
```

181

181

Généricité/polymorphisme statique

182

182

Fonctions *template* (généricité)

- Contexte : Fonctions opérant des opérations similaires sur des types différents
  - exemple de la fonction min
    - int myMin(int,int)
    - double myMin(double,double)
  - corps des fonctions identiques, seul le type des arguments diffère
  - copier-coller ou macro en C-ANSI

183

183

Généricité statique en C (ou en C++)

- 1ère solution : Utilisation de macros
  - Utilisation des possibilités offertes par le précompilateur
    #define min(a,b) (a<b)?a:b
  - Inconvénient : Il n'y a pas de fonction créée, juste de la substitution de code à chaque appel de la macro, sans contrôle sur la cohérence de type des différents paramètres

184

184

tri_tab_macros.H

```
#ifndef _TRI_TAB_MACROS
#define _TRI_TAB_MACROS

#define TRI_TAB(tab,taille,TYPE) \
{ int i, j, indmin;\
  TYPE temp;\
  for (i=0;i<taille-1;i++)\
    { indmin=i;\
      for(j=i+1;j<taille;j++)\
        if(tab[j]<tab[indmin])\
          indmin=j;\
      if(indmin!=i)\
            { temp=tab[indmin];\
              tab[indmin]=tab[i];\
              tab[i]=temp;\
            }\
    }\
}\
//Preconditions : tab[i] initialises
//Postconditions : tab[i]<=tab[i+1]
#endif
```

main.C

```
#include "tri_tab_macros.H"
#include <cstdio>

int main()
{
  double tabd[]={3.4,71.5,2.0,15.5,98.8};
  int tabi[]={4,1,3};
  TRI_TAB(tabd,5,double);
  TRI_TAB(tabi,3,int);
  return 0;
}
```

185

185

- Possibilité C++ de définir une **famille de fonctions** qui ne diffèrent que par le type des arguments manipulés
- Définition d'une famille paramétrée par un nom de type
  → définition d'un patron de fonction (fonction template)

186

186

- Syntaxe d'un patron de fonction :
  - Définition du ou des paramètres génériques
    template <typename T> // Ici, paramétrisation
    // par un nom de type
  - Corps commun à toutes les fonctions de la famille
    T myMin(T e1,T e2)
    {
    return (e1<e2)?e1:e2;
    }
  - myMin<int> et myMin<double> sont des fonctions de la famille ainsi définie et peuvent être utilisées dans un programme
  - TO DO en TP : utiliser des références!

187

187

- Un patron peut aussi être paramétré par une valeur

```
template <typename T, unsigned int TAILLE>
T minTab(T tab[])
{
  T res = tab[0];
  for( int i=0 ; i<TAILLE ; i++)
     if ( tab[i] < res )
       res = tab[i];
  return res;
}
```

Ici le fait de mettre la TAILLE en paramètre template, plutôt qu'en argument est bien entendu discutable!

188

188

- Attention! la définition d'un patron de fonction ne correspond pas à la définition d'une fonction
- La génération d'une ou plusieurs fonctions de la famille se fait à la compilation, par instanciation du paramètre générique

```
template <typename T>
T myMin(T e1,T e2)
{return (e1<e2) ? e1:e2;}
...
int i= myMin<int>(24,1);
int i=myMin(5,9); //instanciation implicite
                  // de la fonction myMin<int>
                  // à la compilation
int i=myMin<>(5,9); //Pas de surcharge non template
double d=myMin(9.9,0); //NON (type différents)
```

189

189

```
template <typename T,unsigned int TAILLE>
 T minTab(T tab[])
 {
   T res=tab[0];
   for(int i=0;i<TAILLE;i++)
      if(tab[i]<res)
        res=tab[i];
   return res;
 }

int t[]={1,2};
i=minTab<int,2>(t); //instanciation explicite
```

190

190

- Prérequis sur les paramères d'instanciation d'un template :

```
template <typename T>
T myMin(T e1,T e2)
 {
   return (e1<e2)?e1:e2;
 }
```

– Que se passe-t-il si on appelle myMin sur des instances d'une classe non munis l'opérateur < ?

191

191

- Prérequis sur les paramères d'instanciation d'un template :

```
template <typename T>
T myMin(T e1,T e2)
 {
   return (e1<e2)?e1:e2;
 }
```

– Que se passe-t-il si on appelle myMin sur des instances d'une classe non munis l'opérateur < ?
– Erreur lorsqu'on essaye d'instancier et de compiler l'instanciation de myMin pour cette classe

192

192

- Notion de concept disponible depuis C++20
  - On peut imposer des contraintes sur les paramètres template :

```
template <typename T>
requires std::totally_ordered<T>
T myMin(T e1,T e2)
 {
   return (e1<e2)?e1:e2;
 }
```

  - Le compilateur refusera d'instancier myMin avec des classes ne satisfaisant pas le concept totally_ordered.
  - La bibliothèque standard offre un certain nombre de concepts de base et le programmeur peut en créer de nouveau (cf. cours ultérieur)

193

193

- Concepts offerts par la bibliothèque standard depuis C++20 (concept library)

same_as, derived_from, convertible_to, common_reference_with, common_with, integral, signed_integral, unsigned_integral, floating_point, assignable_from, swappable/swappable_with, destructible, constructible_from, default_initializable, move_constructible, copy_constructible

boolean-testable, equality_comparable/equality_comparable_with, totally_ordered/totally_ordered_with, three_way_comparable/three_way_comparable_with

Movable, copyable, semiregular, regular

invocable/regular_invocable

equivalence_relation

strict_weak_order

194

- Que penser de :

Module1.hpp
```
template <typename T>
T myMin(T,T);
```

Main.cpp
```
#include"Module1.H"
int main()
{int i=myMin(8,7);}
```

Module1.cpp
```
#include"Module1.H"
template <typenameT>
T myMin(T e1,T e2)
{return (e1<e2) ? e1 :e2;}
```


195

- Instanciation d'une fonction :
  nécessairement dans la même unité de compilation (.cpp) que la **définition** du patron de fonction (mécanisme statique)
  ie. besoin d'accéder au patron dans le main!!
- 2 solutions pour gérer la modularité :
  - soit définition du patron dans fichier d'entêtes .hpp (et instanciation à l'utilisation) ☹
  - soit séparation entre .hpp et .cpp avec définition du patron dans fichier d'implantation, suivie d'instanciations explicites

```
template int myMin<int>(int,int);

ou template int myMin<>(int,int);
ou template int myMin(int,int);
```


196

Classe Template

- But : Appliquer aux classes le mécanisme de généricité précédemment décrit pour les fonctions
- Paramétrisation de la définition d'une classe par un type ou par une valeur calculable à la compilation
- Une classe template n'est pas un type mais un patron de type :
  - modèle générique utilisable pour générer toute une famille de classes
  - les classes souhaitées sont obtenues, à la compilation, par **instanciation** des paramètres template

197

- Exemple
  Si on souhaite paramétrer les Complexes par le type de leurs parties réelle et imaginaire :

```
template<typename T>
class Complexe
{public :
      Complexe(T r,T i) : re(r),im(i) {}
      const Complexe<T> & f (const Complexe<T> &);
  private :
      T re;
      T im;
};
Complexe<int> zi(1,2);
Complexe<double> zd(4.4,1);
typedef Complexe<double> D_Complexe;
```


198

- La définition d'une classe template définit une portée préfixée par le nom de la classe
- Les fonctions membres d'une classe template sont des fonctions template
- Elles sont définies dans la portée de cette classe template *

*Il n'en va pas de même des fonctions amies!

199

```
template <typename T>
class Complexe
{public :
    Complexe(T r=T(),T i=T()) : re(r),im(i) {} // 0 par défaut : bof!
    const Complexe<T> & f (const Complexe<T> &);
    friend Complexe<T> operator + <T> (const Complexe<T> &,
                                       const Complexe<T> &));
 private :
          T re;
          T im;
};

template<typename T>
const Complexe<T> &
Complexe<T>::f (const Complexe<T> & c) {…blabla}

template<typename T>
Complexe<T>
operator+ (const Complexe<T> & c1, const Complexe<T> & c2) {…blabla}
```

200

200

- Déclaration d'existence, ou de définition ultérieure d'une classe template
  (utile en cas de dépendance mutuelle avec une autre classe ou fonction)

```
template <typename A>
class UneClasse;
```

- Exemple :

```
template <typename T>
class Complexe;

template <typename T>
Complexe<T> operator + (const Complexe<T> &,
                        const Complexe<T> &));
```

201

201

Arguments génériques d'une classe

- Une classe peut être paramétrée
  - par un type
  - par un type * ou un type &
  - par des constantes arithmétiques (évaluables à la compilation)
  - par des adresses (y compris des pointeurs de fonction)

```
template <typename T,  T  (*MIN) (T,T) >
class Complexe;
```

202

202

- Possibilité de donner une valeur par défaut à un paramètre générique

```
template <class T= int, int i=10>
class Tableau
{
};
```

- Instanciation

```
Tableau<int,4> v1;
Tableau<Tableau<double> > v2;
//Attention espace entre > et > plus obligatoire depuis C++11)!
Tableau v3;
Tableau<int,10.0> //NON, pas de conversion pour
      // l'instanciation des types arithmétiques
Tableau<3> //NON pas en première position …
```

203

203

- L'instanciation est un mécanisme statique :
  - Instanciation d'une classe : possible **uniquement** dans une unité de compilation (.cpp) contenant la **définition** de son patron
  - Instanciation des fonctions membres d'une classe possible uniquement dans la même unité de compilation que leur définition
  - Attention : Instanciation d'une classe et instanciation d'une de ses fonctions membres sont deux choses indépendantes!

204

204

- Que penser de :

Module1.hpp

```
template <typename T>
class LaClasse
{ …
   T f(T,T);
   …
};
```

Main.cpp

```
#include"Module1.H"
int main()
{LaClasse<int> toto;
  int i=toto.f(8,7);
  …
}
```

Module1.cpp

```
#include"Module1.H"

template <typenameT>
T LaClasse<T>::f(T e1,T e2)
{return (e1<e2) ? e1 : e2;}
```

205

205

- 2 solutions pour gérer la modularité :
  - soit définition du patron **et de ses fonctions membres** dans fichier d'entêtes (.hpp) (inclusion du .hpp et instanciation à l'utilisation)
  - soit séparation entre .hpp et .cpp avec définition des fonctions membres dans fichier d'implantation (.cpp), suivie d'instanciations explicites :

```
template class Complexe<int>;  // à la fin de Complexe.C
                // instanciation de la classe Complexe<int>
                // et de toutes ses fonctions membres !!!!!
```


Spécialisation d'une classe template

- Pour donner une définition différente de certaines instantiations d'une classe template :

```
template<>
class Complexe<double>
{// Définition spécialisée
};
```

- La spécialisation peut n'être que partielle :

```
template <int i>
class Tableau <double,i>
{// Spécialisation partielle pour les tableaux de double
};
```


Spécialisation d'une fonction template

```
template <typename T1, typename T2>
T1 monTemplate(T1 a, T2 b)
{return a+b;}
```

- Pour donner une définition différente de certaines instantiations de monTemplate :

```
template< >
int monTemplate<int,double>(int a, double b)
{return a-b;}
```

- Attention : La spécialisation des fonctions ne peut pas être que partielle (pour l'instant)…

```
template<typename T2>
int monTemplate<int,T2>(int a, T2 b)
{return a+1;}
```


Données et fonctions communes à toutes les instances d'une classe

Membres *static*

- Possibilité de définir :
  - des données membres *static* (variables/constantes de classes)
  - des fonctions membres *static* (méthodes de classes)
- Accès :
  - soit à travers une instance,
  - soit directement à partir du nom de la classe

- Abandonnons dans un premier temps les template …
- Donnée membre statique d'une classe :
  - donnée existant à un **unique** exemplaire,
  - **indépendante** des instances de la classe

```
class CC
  {
  public :
    CC() {compteur++;}
    ~CC() {compteur--;}
    static int compteur;
  };
CC c;
std::cout << CC::compteur << c.compteur;
```

- La **déclaration** (dans un .h) d'une donnée membre static **n'entraîne pas** sa **définition**
  - définition **à l'extérieur** de celle de la classe
  - cette définition est supposée **unique** (ie dans le fichier d'implantation .cpp de la classe et non pas dans son interface)

  int CC::compteur=0;
- Exception : Les **constantes de classe** de type **booléen, caractère ou entier** peuvent être définies* dans la définition de la classe

```
class CC
{ static const int AgeCapitaine = 55;
}; //Mais impossibilité d'accéder à &AgeCapitaine
```

*Avec une valeur d'initialisation calculable à la compilation

212

- Fonction membre statique d'une classe
  - fonction membre invocable
    - à travers une instance,
    - ou à travers le nom de la classe
  - une fonction membre statique n'a pas d'argument implicite
    - manipule les données et les fonctions membres statiques de la classe
    - les données et les fonctions membres d'instances locales

213

```
class CC
  {public :
      CC() {compteur++;}
      ~CC() {compteur--;}
     static int nb_instances() {return compteur;}
   private :
      static int compteur;
  };
int CC::compteur=0; //définition + initialisation
CC c;
std::cout << CC::nb_instances()
          << c.nb_instances();
```


214

Quelles déclarations/instructions sont légales?

```
class Exo
{public :
   void non_stat();
   static void stat1(Exo &);
   static void stat2();
   static void stat_const() const;
 private :
   static int classe_var;
   int instance_var;
};
```

```
void Exo::stat1(Exo & arg)
{
   instance_var=1;
   classe_var=2;
   arg.instance_var=3;
   this->instance_var=4;
   stat2();
   non_stat();
   arg.stat2();
   arg.non_stat();
}
```


215

Template et données membres statiques

```
template <class T>
  class Complexe
  {
  …
   static int compteur;
   static T zero;
  };

  template <class T>
  int Complexe<T>::compteur=0;

  template <class T>
  T Complexe<T>::zero=T();
```


216

Template

- Pour générer du code à la compilation, par instanciation de paramètres template
- Ce code peut concerner la définition :
  - De fonctions
  - De classes
  - De variables ou de constantes globales
  - Ou bien même de typedef sur d'autres types existants!

  En gros tout ce qu'on peut trouver dans une classe!

217

Ingrédients définis dans la portée d'une classe (template ou non)

- Des données membres
- Des fonctions membres (ou méthodes)
- Des noms de type (typedef)

```
class LaClasse
{public :
   typedef int myInt;
};
LaClasse::myInt i=3;
```

218

218

En JAVA

- Il y a t-il des templates en JAVA?

219

219

En JAVA

- Il y a t-il des *templates* en JAVA?
  - Java a voulu se doter d'un système de *template* comme C++
  - Attention :
    - Ce système porte le nom de *template* mais il ne procède pas du tout du même mécanisme que C++
    - Choix différent préservant la compatibilité du *byte code*

220

220

En JAVA

- Template de JAVA
  - Inspiration par Smalltalk
  - Toutes les classes héritent obligatoirement de la classe Object
  - Les *templates* de Java reviennent à manipuler les éléments dont le type est générique, comme des Objects au sens large du terme, et à utiliser l'instanciation par un type pour permettre le *down-cast* à l'utilisation
  - L'intérêt de ces *templates* est que le *down-cast* est transparent à l'utilisateur ☺

221

221

En JAVA

- Template de JAVA
  - Exemple d'une Pile générique avant Java 5
    - Utilisation d'une pile d'Object

```
Stack st=new Stack();
st.push( "Je suis une chaine");
…
Iterator it=st.iterator();
for( ;it.hasNext(); )
     System.out.println((String)it.next());
```

    - Le *down-cast* était géré par l'utilisateur
    - Pourrait-être automatisé si on impose un type dynamique similaire à tous les objets de la Stack!!!

222

222

En JAVA

- Template de JAVA
  - En utilisant les *templates* plus besoin de faire les *down-cast*, et il y a un contrôle sur l'homogénéité de type des objets insérés dans la Stack!
  - Exemple d'une Pile générique avec les *templates*
    - Utilisation d'une pile d'Object de type dynamique String

```
Stack<String> st=new Stack<String>();
st.push("Je suis obligee d'etre une chaine");
...
Iterator<String> it=st.iterator();
for( ;it.hasNext(); )
     System.out.println(it.next());
```

    - Le down-cast en String est réalisé par le *template*, et n'est plus à la charge de l'utilisateur!

223

223

En JAVA

- Template de JAVA
  - Template stack :

```
public class Stack<E> extends Vector<E>{
 …
 public E peek(){…}
 public E pop(){…}
 public void push(E e){…}
 …
}
```

224

224

En JAVA

- Template de JAVA
  - On peut même imposer des contraintes sur le type template ☺

```
public class LeJavaTemplate<E extends LaJavaClasse
& LaJavaInterface1 & LaJavaInterface2>
extends Vector<E>{
 …
 public E methode1(){…}
 public <E2 extends E> E2 methode2(E2 t){…}
 …
}
```

225

225

Quid des templates et des fonctions membres virtuelles?

- Peut-on avoir des fonctions membres virtuelles dans les classes templates?

```
template <typename T>
class LaClassePolymorphe
{public :
   virtual void fonction(T);
   virtual ~LaClassePolymorphe(){}
};
```

226

226

Quid des templates et des fonctions membres virtuelles?

- Peut-on avoir des fonctions membres virtuelles dans les classes templates?

```
template <typename T>
class LaClassePolymorphe
{public :
   virtual void fonction(T);
   virtual ~ LaClassePolymorphe(){}
};
```

- OUI et il y aura une table des fonctions virtuelles pour chaque instanciation de la classe.

227

227

Quid des templates et des fonctions membres virtuelles?

- Peut-on avoir des fonctions membres virtuelles template dans une classe?

```
class LaClassePolymorphe
{public :
   template <typename T>
   virtual void fonction(T);
   virtual ~ LaClassePolymorphe(){}
};
```

228

228

Quid des templates et des fonctions membres virtuelles?

- Peut-on avoir des fonctions membres virtuelles template dans une classe?

```
class LaClassePolymorphe
{public :
   template <typename T>
   virtual void fonction(T);
   virtual ~ LaClassePolymorphe(){}
};
```

- NON : au moment de la création de LaClassePolymorphe on a besoin de connaître toutes les fonctions membres virtuelles pour pouvoir créer la table des fonctions membres virtuelles!

229

229