Introduction au C++

Templates

Pierre-Édouard Cailliau & Nicolas Gazères

Dassault Systèmes
Science & Corporate Research

Référence : « C++ Templates : a complete guide », Vandevoorde & Josuttis (2005), Addison-Wesley.

Présentation du problème (1)

max.cpp

```
int max ( int a, int b) {
  return ( a<b ? b : a );
}

float max ( float a, float b) {
  return ( a<b ? b : a );
}

double max ( double a, double b) {
  return ( a<b ? b : a );
}

MyType max ( MyType a, MyType b) {
  return ( a<b ? b : a );
}
```

- Quel type informatique pour les paramètres de la fonction *max()* ?
  - *int ?*
  - *float ?*
  - *double ?*
  - *MyType ?*
    - type avec ordre
- La *logique* de la fonction *max()* est la même quel que soit le type de données.
- Solution classique: les macros
- **Une meilleure solution**
  - templatiser la fonction.

Présentation du problème (2)

complexe.h

```
class complexe {
public:
  // Constructeurs
  complexe( );
  complexe( MyType re, MyType im );

  // Opérateurs
  complexe operator+( const complexe & ) ;
  bool operator==( const complexe & ) ;

  // Accesseurs
 MyType getReal() const;
 MyType getImag() const;

private:
  MyType _re;
  MyType _im;
};
```

- Quel type informatique pour l'état interne de la classe ?
  - *float ?*
  - *double ?*
  - *MyType ?*
- La *logique* de la classe complexe est la même quel que soit le type de données.
- **La solution**
  - templatiser le type.

Finalités des templates

- Utilité pratique
  - Une forme de factorisation de code
    - Des "macros" très évoluées
    - tirant parti du contrôle statique de type (ie. à la compilation)
  - Une seule mise au point
    - Moins de code, moins de maintenance
- Utilité conceptuelle
  - Programmation générique : une autre forme de polymorphisme
    - un polymorphisme *build-time*
    - orienté performances
- Application: Standard Template Library (STL)
  - Containers & Algorithmes *génériques*
    - *Factoriser une logique*
    - *Le type concerné n'est pas figé*

Les formes principales de templates

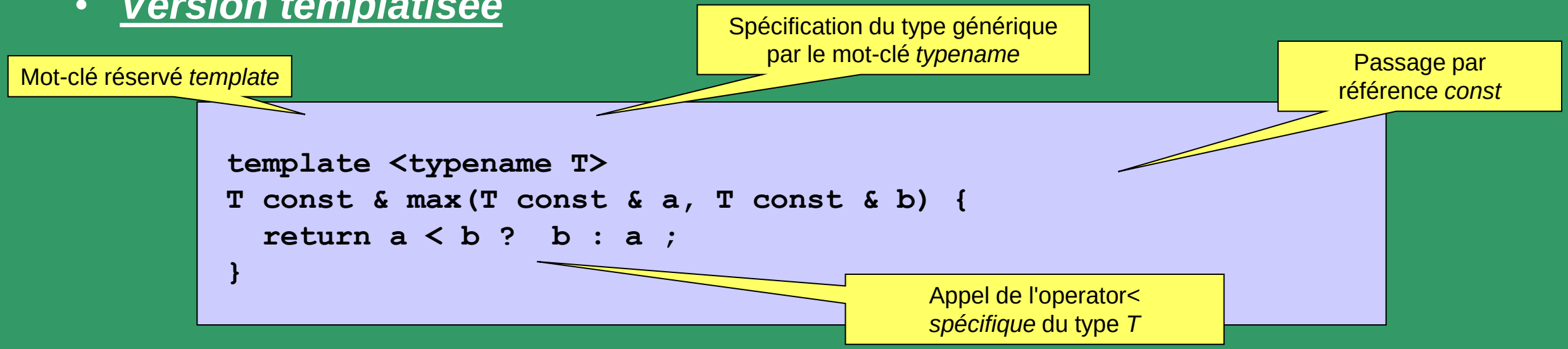
- Templates de fonction
- Templates de classes
- Templates de méthodes ou d'opérateurs

Template de fonction

- Exemple

```
   T    max (   T     a,  T b) {
  return ( a<b ? b : a );
}
```

- ***Version templatisée***



- **Remarque importante**: il y a contrôle de type.
  - Chaque type pour lequel le template est instancié doit posséder un ***operator<***
    - sinon, il y a **erreur de compilation !**

Template de Classe

Mots-clés réservés
template et *typename*

complexe.h

```
template < typename T >
class complexe {
public:
  // Constructeurs
  complexe( );
  complexe(T const & re, T const & im );

  // Opérateurs
  bool operator==(complexe<T> const & );

  // Accesseurs
  T const & getReal() const;
  T const & getImag() const;

private:
  T  _re;
  T  _im;
};
```

complexe.cpp

Il faut rappeler
template<typename T>
dans l'implémentation.

```
template <typename T>
complexe<T>::complexe() {}

template <typename T>
complexe<T>::complexe(T const & re, T const & im )
  : _re(re), _im(im) {
}

template <typename T>
T const & complexe<T>::getReal() const {
  return _re;
}

template <typename T>
bool complexe<T>:: operator==(complexe<T> const & other ) {
  return _re==other._re && _im==other._im ;
}

…
```

Toute occurrence du symbole de la classe template doit faire appraitre le nom de type T.

- **Important**
  - Du template, le compilateur ne génère que le code qui est effectivement utilisé !

Template de méthode

Cas n°1: la classe est une classe ordinaire

- Définition
  - Une méthode template se définit comme un template de fonction dans le scope d'une classe.

MaClasse.h

```
class MaClasse {
public:
 …
   // méthode template
   template<typename U>
   int maMethode(Stack<U> const &);
   …
};
```

MaClasse.cpp

```
template <typename U>
MaClasse::maMethode(U const &) {
  …
}
```

main.cpp

```
int main() {

  MyClass c;

  // Instancie maMethode(int const &)
  c.maMethode( 1 );

  // Instancie maMethode(double const &)
  c.maMethode( 1.0 );

 // Instancie maMethode(char const &)
  c.maMethode( 'a' );

  …
}
```

Template de méthode

Cas n°2: la classe est elle-même un template.

- Définition
  - Template de fonction dans le scope d'une classe qui est elle-même un template.
- Exemple:
  - Opérateur de conversion depuis une Stack dont le type d'objets contenus est différent:

Stack.h

```
// template de classe
template <typename T>
class Stack {
public:
   …
   // template de méthode
   template<typename U>
   Stack<T> & operator=(Stack<U> const &);
};
```

Stack.cpp

```
// définition de la méthode template
// dans la classe template
template <typename T>
  template <typename U>
Stack<T> &
  Stack<T>::operator=(Stack<U> const &)
{
  …;
}
```

Template avec un paramètre-valeur

- C'est une *valeur* (et non-pas un *type*) qui spécialise le template à la compilation
- Exemples

Stack.h

```
template < typename T, int SIZE >
class Stack {
private:
   T elems[SIZE];
   int count;
public:
   …
};
```

addValue.cpp

```
template < int VALUE >
int addValue( int x ) {
   return x+VALUE;
}
```

main.cpp

```
#include "Stack.h"
#include "addValue.h"

int main() {
  …
  Stack< int,48 > s1;
  …
  addValue< 5 >( 7 );
  …
}
```

Exemple important de templates

Les Containers

- Exemple:
  - Stack

```
template <typename T>
class Stack {
   …
public:
   Stack();
   Stack( const Stack<T> & );
   Stack<T>& operator=(Stack<T> const &);

   T top() const;
   …
};

// définition de méthode
template <typename T>
T Stack<T>::top() { … };
```

Instanciation et déduction

Pierre-Édouard Cailliau

École Centrale Paris / OMA

Instanciation d'un template

- Définition
  - L'instanciation d'un template est la génération à la compilation du code du template *pour un type donné*.
- L'instanciation peut être implicite ou forcée:
  - Implicite: **processus de déduction**
    - La simple *utilisation* du template pour un type donné déclenche son instanciation.

```
max( 2.0, 7.0 );              // max<double>,     par déduction
max( 2, 7 );                  // max<int>,        par déduction
```

  - Forcée
    - **Coercition simple**
      - Le programmeur *impose* l'instanciation du template
      - mais le programmeur laisse le compilateur trouver le type *T*.
    - **Coercition complète**
      - Le programmeur *impose* l'instanciation du template et du type *T*.

```
max<>( 7, 42 );               // max<int>, par coercition simple
max<>( 'a', 'b' );            // max<char>, par coercition simple

max<double>( 7, 42 );         // max<double>, par coercition complète
max<int>( 'a', 'b' );         // max<int>, par coercition complète
```

Surcharge d'un nom de fonction par une fonction *standard* et un *template*

- Exemple courant de coexistence:

Max.h

```
// Fonction ordinaire, max pour deux int
int max( int a, int b)
{
  return a<b ?  b : a ;
};


// Template de fonction, à deux paramètres
template <typename T>
const T & max(T const & a, T const & b)
{
  return a<b ?  b : a ;
};
```

L'appel de la fonction ordinaire convient, moyennant une conversion implicite et une troncature.

les conversions implicites ne sont pas prises en compte pour la déduction, donc aucun template ne convient.

Max.cpp

```
max( 2.0, 7.0 );
// Template max<double>, par déduction

max( 'a', 'b' );
// Template max<char>, par déductio
// Entre une conversion implicite e
// correspondance parfaite, le compilateur
// préfère la correspondance parfaite

max( 7, 42 );
// Fonction ordinaire.
// Entre deux correspondances parfaites,
// la forme non-template est préférée.

max<>( 7, 42 );
// Template, par coercition,
// puis max<int>, par déduction.

max<double>( 7, 42 );
// Template max<double>, par coercition.

max( 'a', 42.7 );
// Aucun template ne convient
// Fonction ordinaire, avec
// conversion implicite (pour 'a'),
// et émission d'un warning (pour 42.7) !
```

Le template sait générer une correspondance exacte.

L'appel de la fonction ordinaire requiert une conversion implicite.

Le template sait générer une correspondance exacte.

Il y a correspondance parfaite à la fois pour la fonction ordinaire et le template.

Le développeur force le passage par template.

Le développeur force le passage par template<double>

Le processus de déduction (1)

- On recherche une correspondance exacte de tous les paramètres.
  - Pas de conversion de type implicite
- Exemple

```
template <typename T>
void Fonction1(T const & a, T const & b) {
   …
};
```

```
template <typename T1, typename T2>
void Fonction2(T1 const & a, T2 const & b) {
   …
};
```

```
// OK, correspondance exacte
Fonction1( 2.0, 3.0 );      // Fonction1<double>
Fonction1( 'a', 'b' );      // Fonction1<char>
Fonction1( 13, 14 );        // Fonction1<int>
Fonction1( "abc", "def" );  // Fonction1<char[4]>

// Pas de correspondance
Fonction1( 2.0, 3 );        // double ≠ int
Fonction1( 'a', 65 );       // char ≠ int
Fonction1( "ab", "cde" );   // char[3] ≠ char[4]
```

```
// OK, correspondance exacte
Fonction2( 2.0, 3.0 );     // Fonction2<double, double>
Fonction2( 'a', 'b' );     // Fonction2<char, char>
Fonction2( 13, 14 );       // Fonction2<int, int>
Fonction2( "abc", "def" ); // Fonction2<char[4], char[4]>

// OK, correspondance exacte
Fonction2( 2.0, 3 );       // Fonction2<double, int>
Fonction2( 'a', 65 );      // Fonction2<char, int>
Fonction2( "ab", "cde" );  // Fonction2<char[3], char[4]>
```

Le processus de déduction (2)

- La valeur de retour n'entre pas en jeu.
- Exemple

```
template <typename TARG, typename TRET>
TRET MaFonction(TARG const & a) {
   …
};
```

```
// Ne matche pas
MaFonction( 2.0 );

// OK… mais lourd
MaFonction<double>( 3.1 );
```

Le processus de déduction (3)

- On peut combiner
  - coercition (explicite) pour les premiers paramètres, et
  - déduction (implicite) pour les derniers.
- Exemple

```
template <typename RETOUR, typename ARG1, typename ARG2>
RETOUR MaFonction(ARG1 const & a, ARG2 const & b) {
    …
};


MaFonction<double>( 2.0, 3.1 );    // OK
```

Matche RETOUR=double explicitement
ARG1=double, ARG2=double par déduction.

Le processus de déduction (4)

- Lorsqu'il y a correspondance exacte, une forme non-template l’emporte sur une forme template.

Le processus de déduction (5)

- Comportement intuitif
  - « c'est le template le plus spécialisé qui est choisi ».
- Des règles sophistiquées ont été spécifiées pour garantir ce comportement intuitif.
- Exemple

```
// Template 1
template <typename T>
void Multiply(T const & a, T const & b) {
   …
};

// Template 2
template <typename T>
void Multiply(Array<T> const & a, Array<T> const & b) {
   …
};

void test(Array<int> const & r1, Array<int> const & r2 ) {
  Multiply( r1, r2 );     // La version 2 est choisie → T=int
}
```

Le processus de déduction (6)

- La déduction est récursive
  - Exemple

```
template <typename T, int N> void f(T (&)[N]);

void g() {
  bool b[42];
  f( b );      // → T=bool, N=42
};
```

Instanciation d'un template

remarque

- Ce qui n'est pas utilisé n'est pas instancié

- En particulier:
  - on ne peut pas instancier toute une classe template à la fois
  - ce sont les appels de méthodes qui font l'instanciation

Spécialisation d'un template

- Le principe
  - pour un type donné, imposer une définition particulière du template.
  - La spécialisation peut être complètement différente de la définition originelle
  - Le seul point commun est le nom.
- Exemple

Syntaxe réservée *template<>*

Spécification explicite du type

```
template<typename T>
class S {
  public:
     void info() {
        cout << « I'm generic »;
     }
};
```

```
template <>
class S<void> {
  public:
     void surprise() {
        cout << « I'm special »;
     }
};
```

Utilisations possibles

- Smart pointers
- Holders
  - exclusive ownership buffers
- STL
  - Collections homogènes
- Sérialisation
- Gestion de transactions
  - plusieurs types informatiques à gérer
- Allocateurs génériques
- Compile-time assertions

Comparaison avec les macros

- Les templates sont compilés deux fois
  - Une fois pour la syntaxe (avant l'instanciation)
  - Une fois à chaque instanciation
    - On vérifie que chaque type fourni au template possède les opérations utilisées par le template → vérification statique.
  - Le *#define* d'une macro n'est pas compilé.
- Exemple:
  - Complex et operator<

Comparaison avec le polymorphisme classique

- Polymorphisme des méthodes virtuelles
  - Le **même code** compilé gère une variété de types concrets via une indirection run-time.
  - Contrat = dérivation d'une interface commune
  - Application : containers hétérogènes
- Polymorphisme des templates
  - Le même source gère une variété de types concrets
  - Contrat = le type fourni en argument doit exposer toutes les opérations exploitées par le template.
  - Des **codes spécifiques** sont générés au build-time pour chaque instanciation du template
  - Application : containers homogènes

Templates et Modularisation

Pierre-Édouard Cailliau

École Centrale Paris / OMA

Modularisation naïve

- La modularisation classique entre
  - la déclaration dans un fichier *header*, et
  - la définition dans un fichier *source*

  ne fonctionne plus dès qu'il s'agit de compiler un *template* .

Stack.h

```
template <typename T>
class Stack {
private:
  T * _first;
public:
  Stack( int sz );
  …
};
```

main.cpp

```
#include "Stack.h"

int main() {
  …
  Stack<int> s1;  // Instanciation
  …
}
```

Stack.cpp

```
template <typename T>
Stack<T>:: Stack( int sz ) {
  _first = new T[sz];
};
```

- Que se passe-t'il ?

  → Il y a un symbole irrésolu à l'édition des liens (*link*).

  Explication : l'implémentation du template est compilée séparément de son point d'instanciation.

Solution 1 : le *modèle d'inclusion*

- Le modèle d'inclusion consiste à placer la *déclaration* et la *définition* du template dans le *même* fichier *header*.

Stack_impl.h

```
template <typename T>
Stack<T>:: Stack( int sz ) {
  _first = new T[sz];
};
```

Stack.h

```
template <typename T>
class Stack {
private:
  T * _first;
public:
  Stack( int sz );
  …
};

#include "Stack_impl.h"
```

main.cpp

```
#include "Stack.h"

int main() {
  …
  Stack<int> s1;  // Instanciation
  …
}
```

- Tout fichier qui instancie le template inclut à la fois sa déclaration et son implémentation.
  → Il n'y a plus de symbole irrésolu.
- Mais, que pourrait-il se passer ?

  → Risque de duplicate symbol (au *link*).

  Heureusement : le linker arrive à détecter les doublons et ne garde qu'une seule instanciation de chaque template.

Solution 2 : *l'instanciation explicite*

- L'instanciation explicite consiste à forcer l'instanciation d'un template, pour les types voulus, dans un fichier *source* particulier.

Stack.h

```
template <typename T>
class Stack {
private:
  T * _first;
public:
  Stack( int sz );
  …
};
```

Stack_inst.h

```
template class Stack<int>;
```

main.cpp

```
#include "Stack.h"

int main() {
  Stack<int> s1;  // Instanciation
  …
}
```

Stack.cpp

```
template <typename T>
Stack<T>:: Stack( int sz ) {
  _first = new T[sz];
};

#include "Stack_inst.h"
```

- Chaque template utilisé est instancié au moins une fois manuellement par le développeur.
  → Il n'y a plus de symbole irrésolu.
- Inconvénient ?
  → Nécessité pour le développeur de gérer lui-même ses instanciations.

Compile-time assertions

(Klaus Wittlich, C/C++ Users Journal, Dec. 2005)

- Le besoin
  - Vérifier au build-time qu'une valeur est dans une enum
- La solution

```
// déclaration de template de classe
template<bool> class OnlyTrue;

// spécialisation pour une valeur
template<> class OnlyTrue<true> {}

// Conclusion:
//   sizeof(OnlyTrue<true>) compile
//   sizeof(OnlyTrue<false>) ne compile pas

#define MUSTBETRUE(expr) sizeof(OnlyTrue<expr>)

// Pour vérifier au build-time que x appartient à l'enum
enum State { INPUT, OUTPUT, STREAM } ;
MUST_BE_TRUE(x==State::INPUT || x==State::STREAM || x==State::OUTPUT )
```

Application avancée (1)

- Virtualité paramétrée
  - via une classe de base template
  - la virtualité d'une méthode est déterminée par sa déclaration en *virtual* dans une classe de base.

```
class NonVirtuelle {
};

class Virtuelle {
public:
  virtual void toto() {};
};

template<typename T>
class Derivee : public T {
public:
  void toto() {};
};

// Derivee<Virtuelle>     → toto() est virtuelle
// Derivee<NonVirtuelle>  → toto() n'est pas virtuelle
```

Quizz

- *Déclaration*, *instanciation* ou *spécialisation explicite* ?

- template<>
  class MyClass<float>
  → *spécialisation explicite*

- template<typename T>
  class MyClass ;
  → *declaration*

- template
  class MyClass<float>;
  → *instanciation explicite*