

## PARTIE IV

# Métaprogrammation par les génériques

Bruno Bachelet

Luc Touraille

Christophe Duhamel

# Métaprogrammation par les génériques (1/2)

---

- Repose sur plusieurs mécanismes
- Généricité / patrons de composants
  - En C++: les *templates*
- Instanciation des patrons sans surcoût
  - Au plus proche d'un code dédié
  - En C++: chaque instantiation d'un générique  $\Rightarrow$  code dédié
  - En Java: mécanisme de «*type erasure*»  $\Rightarrow$  pas de code dédié
- Spécialisation «statique»
  - Possibilité de spécialisation d'un composant générique
  - Basée sur le type (ou la valeur statique) des paramètres génériques

# Métaprogrammation par les génériques (2/2)

---

- Objectifs
  - Composants génériques (au sens large)
  - Sans ou avec peu de baisse de performance
- Le génie logiciel nous apprend que généralement
  - La générericité (au sens large) a un coût
  - Recherche d'un compromis entre générericité et efficacité
- Mais avec la programmation générique
  - Beaucoup de travail à la compilation ⇒ peu de pertes

# Quelques utilisations possibles

---

- Evaluation partielle
  - Optimisation de calculs
  - Effectuer une partie d'un calcul à la compilation
- Classes de traits ou politiques
  - Fournir des informations sur les types
  - Ajout non invasif de propriétés ou fonctionnalités
- Métafonctions
  - Fonctions statiques qui produisent du code dynamique
  - Ecrire des algorithmes statiques
- Structures de types
  - Stockage de types (au lieu de données)
  - Manipuler des ensembles de types
- Patrons d'expressions
  - Représentation d'une expression sous forme d'objets
  - Définition d'un langage spécifique embarqué dans C++

- Calcul = partie statique + partie dynamique
- C++ = langage à 2 niveaux
  - Code dynamique: compilé puis exécuté
  - Code statique: interprété à la compilation
    - Basé sur les *templates* + des mécanismes statiques
    - Turing-complet  $\Rightarrow$  pas de limite théorique au niveau algorithmique
- Programmation générique  $\Rightarrow$  évaluation statique d'un calcul
  - L'évaluation statique est généralement partielle
  - On cherchera à effectuer le maximum de calcul statiquement
- Efficacité accrue à l'exécution
  - Peut être très supérieure à l'évaluation dynamique

- Exemple: factorielle  $n!$

- Version dynamique

```
long factorielle(int n) {  
    long r = 1;  
    for (int i = 2; i <= n; i++) r *= i;  
    return r;  
}
```

- Tout se passe à l'exécution

- `y = factorielle(5);`

- Paramètre statique  $\Rightarrow$  possibilité de calcul à la compilation

# Evaluation partielle (3/6)

---

- Version statique, avec fonction générique

```
template <int N> inline long factorielle(void)
{ return (N * factorielle<N-1>()); }
```

```
template <> inline long factorielle<0>(void)
{ return 1; }
```

- Développement du calcul à la compilation

- `y = factorielle<5>();` ⇒ `y = 120;`

- Défauts de cette solution

- Instanciation partielle interdite
  - L'*Inlining* peut être refusé
    - Exemple: code de la fonction trop long

# Evaluation partielle (4/6)

---

- Version statique, avec classe générique

```
template <int N> class Factorielle {  
    public: static const long valeur  
        = N * Factorielle<N-1>::valeur;  
};
```

```
template <> class Factorielle<0> {  
    public: static const long valeur = 1;  
};
```

- Développement du calcul à la compilation

- `y = Factorielle<5>::valeur`  $\Rightarrow$  `y = 120;`

- Avantages de cette solution

- Instanciation partielle possible
  - Pas de problème d'*Inlining*

- Conseil: privilégier l'utilisation d'une structure

- Membres et héritage publics par défaut  $\Rightarrow$  syntaxe allégée

- Exemple d'évaluation partielle: puissance  $x^n$

- Version dynamique

```
double puissance(double x,int n) {  
    double r = 1;  
    for (int i = 1; i <= n; ++i) r *= x;  
    return r;  
}
```

- $n$  est souvent une valeur statique  
⇒ évaluation statique d'une partie du calcul

# Evaluation partielle (6/6)

---

- Version partiellement statique

```
template <int N> struct Puissance {  
    static double calculer(double x)  
    { return (Puissance<N-1>::calculer(x) * x); }  
};  
  
template <> struct Puissance<0> {  
    static double calculer(double) { return 1.0; }  
};
```

- Une partie du développement du calcul à la compilation

- `y = Puissance<5>::calculer(1.2);`  
⇒ `y = puissance_5(1.2);`
  - `double puissance_5(double x)`  
`{ return x*x*x*x*x; }`

# Classes de traits ou politiques (1/4)

---

- Ajout non invasif de propriétés ou fonctionnalités à un type
- Trait = propriété / caractéristique
  - Attribut statique ou type interne
- Politique = fonctionnalité / comportement
  - Méthode statique
- Permet de conserver une indépendance vis-à-vis d'un type
- Tout en produisant un code dédié
  - Classe de traits ⇒ informations spécifiques
  - Classe de politiques ⇒ comportements adaptés

# Classes de traits ou politiques (2/4)

---

- Exemple: savoir si un type représente un entier

- Définition d'une classe de traits

```
template <typename T> struct EstEntier {  
    static const bool valeur = false;  
};
```

```
template <> struct EstEntier<int> {  
    static const bool valeur = true;  
};
```

```
template <> struct EstEntier<long> {  
    static const bool valeur = true;  
};
```

- Utilisation de la classe de traits

```
if (EstEntier<X>::valeur) // code 1  
else // code 2
```

# Classes de traits ou politiques (3/4)

---

- Problématique: afficher le contenu d'un vecteur

```
template <typename T>
```

```
void afficher(const vector<T> & v) {  
    for (unsigned i = 0; i < v.size(); ++i)  
        cout << v[i] << " ";  
}
```

- Le résultat n'est pas forcément celui attendu

- $T = \text{int} \Rightarrow$  affichage entiers
  - $T = \text{int}^* \Rightarrow$  affichage pointeurs

- Objectif: afficher des valeurs qu'il y ait indirection ou non
  - La politique d'accès diffère suivant la nature de  $T$
  - Proposition d'une classe de politiques

# Classes de traits ou politiques (4/4)

---

- Définition de la classe de politiques

```
template <typename T> struct AccesLecture {  
    static const T & getValeur(const T & v) { return v; }  
    static const T * getPointeur(const T & v) { return &v; }  
};  
  
template <typename T> struct AccesLecture<T *> {  
    static const T & getValeur(const T * p) { return *p; }  
    static const T * getPointeur(const T * p) { return p; }  
};
```

- Utilisation de la classe de politiques

```
template <typename T>  
void afficher(const vector<T> & v) {  
    for (unsigned i = 0; i < v.size(); ++i)  
        cout << AccesLecture<T>::getValeur(v[i]) << " ";  
}
```

# Métafonctions (1/4)

---

- Métafonction = classe générique agissant comme une fonction
  - Permet l'exécution statique d'un algorithme
- Elément central de la métaprogrammation par génériques
  - Métaprogrammation = code qui génère du code
  - Métafonction = fonction statique qui produit du code dynamique
- Reçoit des paramètres et retourne un résultat
  - Peuvent être des nombres statiques
  - Peuvent être des types
- Nécessité de manipuler types et nombres indifféremment
  - Métadonnée = type ou nombre statique
  - Représentation unifiée par des classes génériques
  - Métadonnée embarquée dans un membre de classe

- Pour les nombres

```
template <typename T,T VAL> struct Nombre {  
    typedef T type;  
    static const T valeur = VAL;  
};
```

- Pour les types

```
template <typename T> struct Type  
{ typedef T type; };
```

- Exemple: condition «si»

```
template <typename TEST,typename SI,typename SINON,  
        bool = TEST::valeur> struct Si : SINON {};
```

```
template <typename TEST,typename SI,typename SINON>  
struct Si<TEST,SI,SINON,true> : SI {};
```

- Remarque: utilisation de l'héritage pour «retourner» le résultat

- Manipulation de nombres

```
template <typename N> struct EstNegatif  
: Nombre< bool,N::valeur<0 > {};
```

```
template <typename N> struct ValeurAbsolue  
: Si< EstNegatif<N>,  
    Nombre<typename N::type,-N::valeur>,  
    Nombre<typename N::type,N::valeur>  
  > {};
```

- Exemple d'utilisation

- `y = ValeurAbsolue< Nombre<int,-5> >::valeur;`

- Manipulation de types

```
class Algo1 { ... public: static void executer(); };  
class Algo2 { ... public: static void executer(); };
```

```
template <typename T> struct MeilleurAlgo  
: Si< EstEntier<T>,  
    Type<Algo1>,  
    Type<Algo2>  
  > {};
```

- Exemple d'utilisation

- **MeilleurAlgo<int>::type::executer()**

# Structures de types (1/3)

---

- Au lieu de stocker des données, stocker des types
  - Exemple connu: les «*typelists*»

- Structure de liste chaînée statique

```
template <typename TYPE,typename SUIVANT>
struct ListeType {
    typedef TYPE element;
    typedef SUIVANT suivant;
};

struct TypeNul {};
```

- Construction d'une liste de types

```
typedef ListeType<int,
                  ListeType<long,
                  ListeType<double,TypeNul>
                >
> types_nombre_t;
```

# Structures de types (2/3)

---

- Recherche d'un type

```
template <typename LISTE,typename TYPE>
struct Contient
: Ou< MemeType<typename LISTE::element,TYPE>,
  Contient<typename LISTE::suivant,TYPE>
> {};
```

```
template <typename TYPE>
struct Contient<TypeNul,TYPE>
: Nombre<bool,false> {};
```

- Exemple d'utilisation

```
if (Contient<types_nombre_t,x>::valeur) // Code 1
else // Code 2
```

# Structures de types (3/3)

---

- Métafonctions nécessaires à l'exemple précédent

- Comparer deux types

```
template <typename T1,typename T2>
struct MemeType : Nombre<bool,false> {};
```

  

```
template <typename T>
struct MemeType<T,T> : Nombre<bool,true> {};
```

- Opérateur «ou»

```
template <typename N1,typename N2> struct Ou
: Nombre<bool,N1::valeur || N2::valeur> {};
```

# Patrons d'expressions (1/4)

---

- Terme anglais: *Expression templates*
- A partir de la surcharge d'opérateurs  
⇒ arbre syntaxique d'une expression
- Objectifs
  - Définir d'un langage spécifique embarqué dans C++
    - EDSL (*Embedded Domain-Specific Language*)
  - Optimiser l'évaluation d'une expression
    - L'idée est de différer un calcul (e.g. **a\*b**)
    - En vue d'optimiser l'expression complète (e.g. **a\*b\*c**)
- Exemple: calcul matriciel

# Patrons d'expressions (2/4)

---

- Surcharge  $\Rightarrow$  objet retourné au lieu du résultat attendu
  - Exemple: **a+b**
  - Retourne un objet: **Addition<A,B>**
- De cette manière, on peut représenter une expression complète
  - Exemple: **a+b\*c**
  - Retourne un objet: **Addition< A,Multiplication<B,C> >**
  - On obtient un arbre syntaxique
- Le calcul peut être optimisé
  - Exemple: **x = a+b\*c**
  - Surcharge de l'opérateur d'affectation  
 $\Rightarrow$  appel méthode «**evaluer**» optimisée de l'expression complète
- Un aperçu (ultra light !)...

# Patrons d'expressions (3/4)

---

- Représentation des opérations

```
template <typename E1,typename E2> class Addition {  
protected: E1 e1;  
protected: E2 e2;  
  
public: Addition(const E1 & a,const E2 & b)  
: e1(a),e2(b) {}  
  
public: double evaluer(void) const  
{ return (e1.evaluer() + e2.evaluer()); }  
};
```

- Calcul différé

```
class Resultat {  
protected: double v;  
  
public: template <typename E>  
Resultat & operator = (const E & e)  
{ v = e.evaluer(); return (*this); }  
};
```

# Patrons d'expressions (4/4)

---

- Représentation des opérandes

```
class Operande {  
protected: double v;  
  
public: Operande(const double & a) : v(a) {}  
  
public: const double & evaluer(void) const { return v; }  
};
```

- Surcharge des opérateurs

```
template <typename E1,typename E2>  
Addition<E1,E2> operator + (const E1 & a,const E2 & b)  
{ return Addition<E1,E2>(a,b); }
```

- Exemple d'utilisation

```
Resultat r;  
r = Operande(3) + Operande(17.2) + Operande(12.7);
```