Université Lyon 1
Master d'Informatique

Programmation Générique et le langage C++

Norme ISO

Raphaëlle Chaine
raphaelle.chaine@liris.cnrs.fr
2012-2013

Meta-programmation avec les template

- Mécanisme des template :
  Génération de code par interprétation à la compilation
- Lorsque le compilateur rencontre du code correspondant à une instantiation d'un template :
  - génération de la spécialisation associée aux paramètres fournis.

– La metaprogrammation consiste à pousser plus loin les possibilités offertes par le compilateur, de manière à générer du code encore plus performant… au prix d'une compilation plus longue!
– Un metaprogramme génère et manipule du code corrrespondant à des programmes pendant leur compilation
– Sens littéral : « Un programme sur les programmes »

- Domaine récent :
  Naissance de nouvelles astuces template de metaprogrammation provoquant l'effervescence des experts
- Quelles en sont leurs applications?
- Comment acquérir les connaissances de base permettant de les utiliser avec discipline?

- Template à base numérique
  - Optimisation des performances requises par des calculs mathématiques
  - Exponentielle, sinus, factorielle, calculs matriciels
- Template à base de types
  - Typelists
  - Implantation de design patterns (fabrique, visiteurs, …) en conservant un typage fort
  - Analyseurs syntaxiques …

Compilateur C++ : une machine de Turing!

– Pour preuve, programme de Erwin Unruh qui calcule les nombres premiers à la compilation, et les affiche à travers les messages d'erreur du compilateur (1994)

```
template <int i> struct D {
     D(void*);
     operator int();
   };
template <int p, int i>
struct is_prime {
    enum { prim = (p==2)
                  ||(p%i)
                   && is_prime<(i>2?p:0),i-1>::prim
     };
   };
template <int i> struct Prime_print {
    Prime_print<i-1> a;
    enum { prim = is_prime<i, i-1>::prim };
    void f() { D<i> d = prim ? 1 : 0; a.f();}
   };
template<> struct is_prime<0,0> { enum {prim=1}; };
template<> struct is_prime<0,1> { enum {prim=1}; };
template<> struct Prime_print<1> {
    enum {prim=0};
    void f() { D<1> d = prim ? 1 : 0; };
   };
#ifndef LAST
#define LAST 18
#endif
main() {
    Prime_print<LAST> a;
    a.f();
   }
```

- Autre illustration par un exemple classique : calcul de la factorielle d'un nombre entier.
- Les énumérations permettent de donner des noms à des valeurs
- Principe :
  - utiliser des noms paramétrés par des valeurs numériques
  - mais on ne peut pas paramétrer une enumération par un paramètre template …
  - En revanche on peut l'encapsuler dans une classe template!

```
template <unsigned int N> struct Factorielle
{
  enum {valeur = N * Factorielle<N-1>::valeur};
};
```

- Cette construction récursive trouve un cas d'arrêt dans une spécialisation du template.

```
template <> struct Factorielle<0>
{
  enum {valeur =1};
}

unsigned int x =Factorielle<3>::valeur;
```

- Dès la compilation x vaudra 6
  (économie du calcul à l'exécution)

Que penser de :

```
unsigned int i=4;
unsigned int x
=Factorielle<i>::valeur;
```

- Autre exemple classique :
  Au lieu d'utiliser des enumérations, on peut aussi utiliser des données membres statiques

```
template <unsigned int N>
struct Binaire
{
  static const unsigned int valeur
     =Binaire<N/10>::valeur *2 + N%10;
};
```

- Spécialisation :

```
template <>
struct Binaire<0>
{
  static const unsigned int valeur = 0;
}
```

- Utilisation :

```
const unsigned int
i=binaire<1001>::valeur;
```

La valeur 9 de i est calculée à la compilation

- Attention :

`binaire<54>::valeur` n'a pas de sens!
Mais il existe des techniques pour s'assurer que le paramètre d'instanciation est bien composé de 0 et de 1 (Abrahams, Gurtovoy)

- Dans un méta programme, une classe template peut être considérée comme une fonction dans sa forme la plus simple : **prenant des paramètres numériques et retournant une (ou plusieurs!) valeurs**

- Attention les paramètres template ne peuvent être des flottants!
  (mais `3.0f/4.0f` calculé à la compilation ☺)

- Metaprogramme fabriquant un flottant :

```
template <int N> inline double Factorielle()
{return N*Factorielle<N-1>();}
template <> inline double Factorielle<0>()
{return 1.0;}
```

- Structure de contrôle dans un meta programme:
  - Quel équivalent du `if/else` dans un metaprogramme?

- Utilisation d'une classe templatée par des booléens et dont les 2 instanciations correspondent à des comportements différents

```
template <bool Condition>
struct Test
  {};
```

- Spécialisations :

```
template <> struct Test<true>
  {
     static void traitement()
        { traitement1();//si possible inline}
  };
template <> struct Test<false>
  {
     static void traitement()
        {traitement2(); //si possible inline}
  };
```

- Des instructions du type

```
if (Condition)
   traitement1();
else
    traitement2();
```

- Pourront ainsi être remplacées par :

```
Test<Condition>::traitement();
```

- A condition que `Condition` soit évaluable à la compilation!!!

- L'opérateur conditionnel ternaire peut être évalué à la compilation
  - Exemple d'utilisation :

```
template <unsigned int I> struct NumTest
{
 enum
 {
    Pair = (I%2? false : true),
    Zero = (I==0 ? true : false)
  };
};
```

  - On peut ensuite utiliser les valeurs booléenne `NumTest<I>::Pair` et `NumTest<I>::Zero` (si I évaluable à la compilation!)

- Boucles dans un meta programme:
  - Quel équivalent du `for` dans un metaprogramme?

- Utilisation d'une construction récursive utilisant une classe templatée par une valeur numérique marquant le démarrage et une valeur numérique marquant la fin de la boucle

```
template <unsigned int Debut, unsigned int fin>
struct Boucle
{
   static void traitement()
   { MonTraitement();
     Boucle<Debut+1,End>::traitement();
   }
};
```

- Spécialisation partielle

```
template < unsigned int N>
struct Boucle<N,N>
{
   static void traitement() {}
};
```

Des instructions du type

```
for (int i=0;i<10;i++)
  MonTraitement();
```

Pourront ainsi être remplacées par :

```
Boucle<0,10>::traitement();
```

Danger si `Debut>Fin`
(peut être vérifié à la compilation)

- Le compilateur permet de réaliser des traitements fonctionnels, mais également de stocker des résultats dans des variables temporaires ☺

```
template <int X, int Y, int Z>
struct MetaProg
  {
     enum
     {
       v1 = NumTest<Z>::Pair;
       v2= X*v1+Y;
       v3 = NumTest<v2>::Zero ? X : 2*Y;
       valeur = v1*v3;
      };
  };
```

Utilisation

- Réécriture optimisée de fonctions mathématiques :
- Fonction puissance :

```
template  <unsigned int N>
inline double Puissance(double x)
  {return x*Puissance<N-1>(x);}
template <>
inline double Puissance<0>(double x)
{return 1.0;}
```

- Ici, seul N est un paramètre template, mais pas x (qui peut donc être non évaluable à la compilation)

```
double x, y;
  …
Y=puissance<7>(x);
```

- Réécriture optimisée de fonctions mathématiques qui s'approximent à partir de leur développement en série entière
  - Si on tronque la série entière à partir d'un degré N, on obtient un polynome de degré N
    (N=précision numérique)
  - Ex : exponentielle, cos, sin, atan, etc…
- Réécriture de calculs vectoriels et matriciels en utilisant des expressions template (cf. boost::uBlas)
  - Idée : éviter la construction d'objets temporaires
  - Construction arborescente de template
  - Paramètres template = iterateurs sur les vecteurs

Les typelists

- Listes de types (présentes dans `boost`)
- Utilisées pour la génération automatique de code

```
template < typename T1, typename T2>
struct TypeList
{
 typedef T1 Head; //premier type
 typedef T2 Tail; //liste des types restants
};
struct NullType {};
```

- Aucune donnée dans une `TypeList`

- A partir de cette structure template, on peut fabriquer des listes de types, selon une construction récursive.
- Exemple :

```
typedef TypeList< int,
    TypeList<double,NullType> > l_int_double;
```

- Simplification de la construction à l'aide de macros :

```
#define TYPELIST_1(t1)
   TypeList<t1, NullType>
#define TYPELIST_2(t1,t2)
   TypeList<t1, TYPELIST_1(t2)>
#define TYPELIST_3(t1,t2,t3)
   TypeList<t1, TYPELIST_2(t2,t3)>
```

- Puis

```
typedef TYPELIST_2(int,double) l_int_double;
```

Opérations sur les typelists

- Méta fonctions:
  - Length<typename TL>
  - IndexOf<typename TL, typename T>
  - TypeAt<typename TL, int N>
  - Union<typename TL1, typename TL2>
  - …

Length<typename TL>

- Spécialisations :

  `template < typename H, typename T>`

  Métafonction `Length< TypeList<H,T> >`

  `… valeur = 1 + Length<T>::valeur`

- `template<>`

  Métafonction `Length<NullType>`

  `… valeur = 0`

- Renvoie une erreur si on n'utilise pas une TypeList

Comme vu précédemment, les métafonctions peuvent être mise en œuvre à l'aide de données membres statiques ou des types énumérés encapsulés dans des structures templates

IndexOf<typename TL, typename A>

- Spécialisations
  `template <typename T, typename A>`
  Métafonction `IndexOf< TypeList<A,T>, A >`
  `… valeur = 0`
- `template <typename A>`
  Métafonction `IndexOf<NullType, A>`
  `… valeur = -1`
- `template <typename H, typename T, typename A>`
  Métafonction `IndexOf< TypeList<H,T>, A >`
  Utilisation d'une variable temporaire
  `temp = IndexOf<T, A> //recherche sur la Queue`
  Si `(temp == -1)` alors `valeur = -1 // pas trouvé`
  Sinon `valeur = 1 + temp // trouvé`

Utilisation

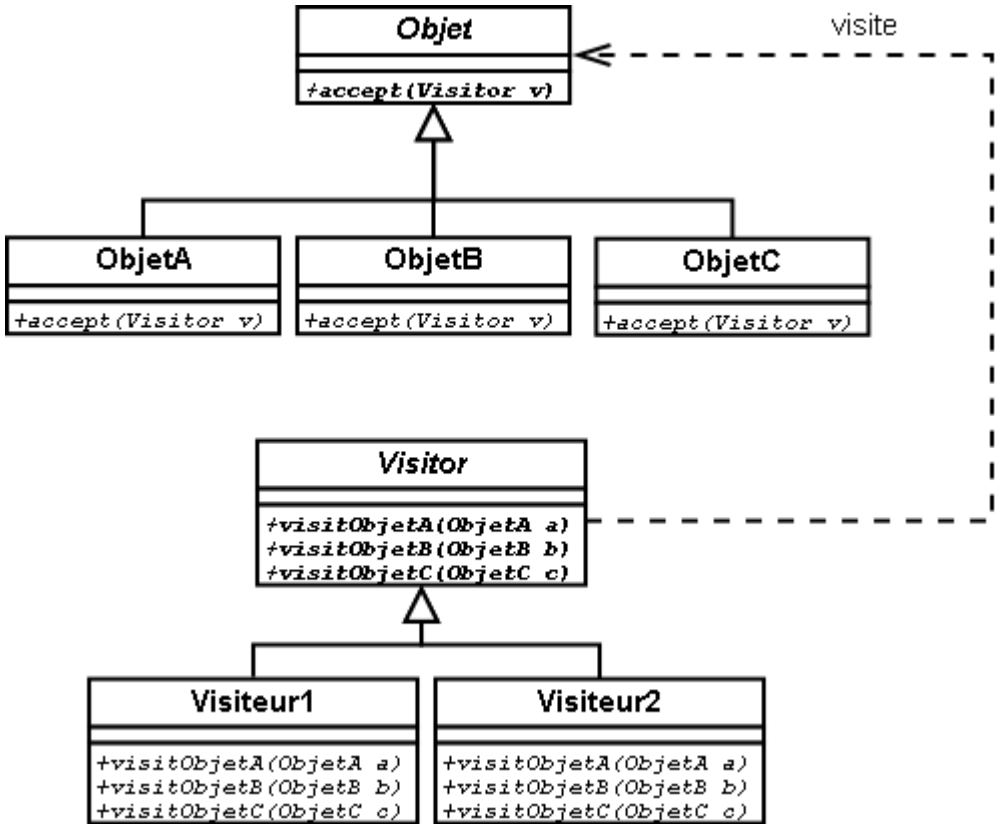
- Permet de définir des familles de type
  - `typedef TYPELIST_4(int, char, long, short) EntierSigné`
  - `typedef TYPELIST_4<uint, uchar, ulong, ushort> EntierNonSigné`
  - `typedef Union<EntierSigné, EntierNonSigné> Entier`
  - `typedef TYPELIST_2<float, double> Flottant`
  - `typedef Union<Entier, Flottant> Nombre`
- Permet d'explorer un ensemble de type (par exemple pour générer du code associé)

Utilisation

- Génération de classes via la génération automatique de hierarchies
- Exemple :
  - Etant donné une hiérarchie de classes
    Ex : Hiérarchie de classe Figure
    - rectangle,
    - rond,
    - groupe de Figures [*pattern composite*]

```
class Figure {  …  };
class groupe : public Figure { …
  set<Figure *> composants;
  };
```

  - On veut mettre en place une deuxième hiérarchie de classe correspondant à un pattern Visiteur
    (ex: pour visualiser)
  - Permet une séparation données/traitement)



http://pcaboche.developpez.com/article/design-patterns/

Pour chaque classe d'objet visité :

- fonction membre **virtuelle** `accept`:

```
void ObjetDeTypeA::accept( Visitor * v )
{ v->visitObjet( this ) ; }
```

Pour chaque classe de visiteur :

- Autant de fonctions membres **virtuelle**s `visitObject` que de classes dans la hiérarchie d'Objets

```
void MonVisiteur::visitObjet(ObjetDeTypeA * o)
  { // Traitement d'un objet de type A }
void MonVisiteur::visitObjet(ObjetDeTypeB * o)
  { // Traitement d'un objet de type B }
void MonVisiteur::visitObjet(ObjetDeTypeC * o)
  { // Traitement d'un objet de type C }
```

- Utilité d'une génération de code automatique :
  Une fonction visitObject par type visité!

- N’écrire le code qu’une seule fois dans un template, puis l’instancier pour chacun des types d’une TypeList
- ```
  template <class OO>
  class PatronVisitor
  {
    void visitObjet(OO *o);
  };
  ```
- La classe visiteur finale dérivera de toutes ses instanciations!
  ```

```
template <class TList,
          template <class> class PVisitor>
class HierarchieVisiteur;
```

Spécialisations :

- ```
  template <class T1, class T2,
            template <class> class PVisitor>
  class HierarchieVisiteur<TypeList<T1,T2>, PVisitor >
   : public PVisitor<T1>,
     public HierarchieVisiteur<T2,PVisitor>
     {}; // 2 héritages
  ```
- ```
  template <class T,
            template <class> class PVisitor>
  class HierarchieVisiteur<TypeList<T,NullType>,
                           PVisitor >
   : public PVisitor<T>
    {}; // 1 seul héritage dans le cas d'une liste
        // contenant un seul type
  ```

Static_assert

- Actuellement présent dans boost
- L'assertion statique est un test à la compilation

```
template <boolTest> StaticAssert;
template<> StaticAssert<true> {}; // vide
```

Evaluation statique de Test

- `true` : le compilateur passe le test
- `false` : erreur `StaticAssert<false>` non défini

Exemple :

- `StaticAssert< sizeof(T) == 4>`
- vérifie que le type T occupe 4 octets

Notions de concept et de modèle

- Lorsqu'on définit un template générique, certaines propriétés syntaxiques mais aussi sémantiques sont attendues de la part des paramètres *template*
- En cas de non respect, à l'instantiation :
  1. d'un prérequis syntaxique :
     erreur de compilation (pas très sibylline) à l'endroit où est réalisé l'appel à la propriété syntaxique défaillante!
     --- On aurait préféré être renseigné dès l'instanciation ---
  2. d'une prérequis sémantique :
     aucune erreur de compilation, mais risque de déroulement incorrect du programme

- Catégorisation des types abstraits de données en concepts décrivant des prérequis syntaxiques et sémantiques
- Concepts permettant de spécifier
  - à quels types un traitement générique est applicable
  - quels types peuvent être substitués aux paramètres template formels

- Un type satisfaisant à un concept C est dit modèle de ce concept C

- Un modèle qui ajoute des prérequis à un autre correspond à un raffinement de ce concept

- Un template doit fournir ses exigences en termes de concepts
- Concepts de la STL :
  `Assignable, Default Constructible, Less than Comparable, Equality Comparable, Container, Iterator, Algorithms, Fonctors, Adaptors,…`
- Dans un concept peuvent aussi apparaître des invariants (associativité, symétrie,…) ou des garanties en termes de complexité.

- Mais aucun contrôle n’est fait à l’instanciation!
- Historiquement, les concepts C++ ne jouent qu’un rôle de documentation
- La responsabilité de la vérification incombe à l’utilisateur!!

Concept GCC

- Prototype de compilateur C++ (construit sur le compilateur GNU C++) supportant des extensions du langage utiles pour la programmation générique
- La notion de concept est matérialisée dans ces extensions et devrait être reprise dans une prochaine normalisation du C++ …
  (pas dans C++11)
  - « Concept checking »
  - Clarification des messages d'erreur
  - Vérification de certaines contraintes sémantiques

```
concept NomConcept < typename T1, typename T4>
  : NomConceptBase<T1>
  {
    typename T2; // type associé
    typename T3 = T1; // avec spécification
                      // par défaut
    T2 fonction(T1,T4);
    where std::Convertible<T2,T1>;
    T1 operator+(T1,T1);
    axiom UneCondition(T1 x, T1 y)
       {x+y ==y+x;} //Invariant
  };
```

Puis :

```
  template <typename T>
  where NomConcept<T,int>
  class Toto ....
```

Décryptage

- Un type T1 et un type T2 constituent un modèle du concept `NomConcept` si
  - T1 est un modèle du concept NomConceptBase
  - Il existe une fonction `fonction` prenant en argument un T1 et un T4 et retournant un T2 (avec la contrainte que T2 et T1 doivent être un modèle du concept `std::convertible`),
  - ...

- Pour déclarer qu'un(e famille de) type(s) T satisfait un concept, utilisation de **concept_map**

```
template <typename T>
  where Float<T>
concept_map NomConcept<T,float>
{
  typedef double T3;
}
// les modèles de float sont aussi
// des modèles de NomConcept
```

- Cette formalisation des concepts devraient remplacer les actuelles classes de traits et classes de politiques
- Classes de traits :
  - informations sur un type ou une constante
  - dans la STL : `numeric_limits<int>::max() = 2^(31)-1`
  - InfoType<T>::Nom() qui fournirait le type de T en toutes lettres
  - InfoType<T>::Zero()
  - Contiennent :
    - Données et fonctions membres statiques,
    - Des typedef
- Ex :

```
template <>
    struct InfoType<Complexe>
    { static inline const Complexe Zero() =
                                        complex(0,0);
      static inline const std::String nom() =
                                        "Complexe";
       typedef  ….
    };
```

- Classes de politiques
  - Par rapport aux traits, les politiques donnent un comportement
  - Exemple classique

```
template <T,
  template <class> class Politique>
T accumulation(T tab [], int n)
  {
    T somme = infoType<T>::Zero();
    for(int i=0; i<n ; i++)
       Politique<T>::incremente(somme,
                                tab[i]);
    return somme;
  }
```