ISIMA

Génériques à paramètres variables

(PARTIE VII – C++11/14)

Bruno Bachelet

Loïc Yon

Syntaxe *(1/2)*

- *Variadic template* = générique à paramètres variables
  - Liste des paramètres *templates* non fixée
  - A l'instar des arguments variables d'une fonction

- Permet de modéliser des collections hétérogènes
  - `template <typename... TYPES> class Tuple;`

- Permet de renforcer le contrôle de types
  - `fprintf(const char * format,...);`
    - Impossible d'identifier les types des arguments variables
  - ```
    template <typename... TYPES>
    void printf(const char * format,TYPES... args);
    ```
    - Possibilité d'identifier le type de chaque argument variable
    ```

Syntaxe *(2/2)*

- Paramètres variables = «pack» de paramètres
  - Pack représenté par le symbole «`...`»
  - Pack = 0 à *n* paramètres
- Pack de valeurs: `template <int... VALEURS>`
- Pack de types: `template <typename... TYPES>`
- Pack de génériques
  - Rappel: `template <template <typename> class T> class X;`
    - Le paramètre *T* de la classe générique *X* est un générique
  - `template <template <typename> class... GENERIQUES>`
    - Pack de génériques à un paramètre
  - `template <template <typename...> class... GENERIQUES>`
    - Pack de génériques à paramètres variables

Expansion de pack *(1/4)*

- `template <typename... TYPES>`
  ⇒ `template <typename T1, … ,typename Tn>`
  - Il s'agit d'une illustration: *T1…Tn* n'existent pas explicitement

- Accès direct à un paramètre d'un pack impossible
  - Un paramètre n'a pas d'identifiant
    - Aucun moyen d'obtenir le nom d'un paramètre
  - Un paramètre n'a pas de numéro
    - Aucun moyen direct d'obtenir le n^{ième} paramètre

- Parcours et identification possibles par récursivité
  - Ecriture de *templates* récursifs ⇒ métaprogrammation

- Nombre d'éléments d'un pack: opérateur `sizeof...(`*PACK*`)`
  - Remarque: un *template* récursif peut aussi faire l'affaire

Expansion de pack *(2/4)*

- Comment utiliser ces paramètres alors ?
- Grâce au mécanisme d'expansion
- Schéma d'expansion
  - Expression contenant l'identifiant d'un pack
  - Et terminée par « . . . »
- Expansion = réplication du schéma
  - Pour chaque paramètre du pack
  - Séparation par une virgule

Expansion de pack *(3/4)*

- Supposons un pack
  - `template <typename... TYPES>`

- Exemples d'expansions possibles
  - `TYPES...`
    ⇒ `T1,` … `,Tn`
  - `X<TYPES>...`
    ⇒ `X<T1>,` … `,X<Tn>`
  - `X<TYPES>::a...`
    ⇒ `X<T1>::a,` … `,X<Tn>::a`
  - `X<TYPES>::m(u,v)...`
    ⇒ `X<T1>::m(u,v),` … `,X<Tn>::m(u,v)`
  - `const TYPES &... x`
    ⇒ `const T1 & x1,` … `,const Tn & xn`

Expansion de pack *(4/4)*

- Expansion de plusieurs packs
  - Expansions simultanées
    - `X<PACK1,PACK2>...` ⇒ `X<T1,U1>,` … `,X<Tn,Un>`
  - Expansions séparées
    - `X<PACK1...,PACK2...>` ⇒ `X<T1,` … `,Tn,U1,` … `,Un>`

- La localisation de «`...`» est importante
  - Exercice: trouver les expansions suivantes (Andrei Alexandrescu)

```
template <typename... TYPES> void f(TYPES... args) {
 g(A<TYPES...>::m(args)...);
 g(A<TYPES...>::m(args...));
 g(A<TYPES>::m(args)...);
}
```

  - Remarque: «`args`» représente un pack d'arguments
    - Mécanisme d'expansion similaire au pack de paramètres

Modélisation d'un tuple *(1/3)*

- Aperçu de «**`std::tuple`**»
- Déclaration du *template*
  - `template <typename... TYPES> class Tuple;`
- Cas d'un pack vide
  - `template <> class Tuple<> {};`
- Cas général (récursion)
  - Pack séparé en deux: le premier paramètre et le reste
  - ```
    template <typename T,typename... RESTE>
    class Tuple<T,RESTE...> : public Tuple<RESTE...> {
     public:
      T value;

      Tuple(const T & val,const RESTE &... args)
      : Tuple<RESTE...>(args...), value(val) {}
    };
    ```

Code source complet: `cpp11_tuple.cpp`
    ```

Modélisation d'un tuple *(2/3)*

- Accès au n^{ème} élément d'un tuple ⇒ métaprogrammation
- Déclaration du *template*
  - `template <unsigned N,typename TUPLE> struct TupleElement;`
- Récursion

```
template <unsigned N,typename T,typename... RESTE>
struct TupleElement< N, Tuple<T,RESTE...> >
: TupleElement< N-1, Tuple<RESTE...> > {};
```

- Condition d'arrêt

```
template <typename T,typename... RESTE>
struct TupleElement< 0, Tuple<T,RESTE...> > {
 typedef T type;

 static T & get(Tuple<T,RESTE...> & t) { return t.value; }
};
```

Modélisation d'un tuple *(3/3)*

- Utilisation du *template*
  - `typedef Tuple<int,double> MyTuple;`
  - `MyTuple tuple(13,2.7);`
  - `TupleElement<1,MyTuple>::type` ⇒ *double*
  - `TupleElement<1,decltype(tuple)>::get(tuple)` ⇒ *2.7*

- Méthode «`get`» peu pratique ⇒ fonction d'assistance

```
template <unsigned N,typename... TYPES>
typename TupleElement< N, Tuple<TYPES...> >::type &
get_value(Tuple<TYPES...> & t) {
 return TupleElement< N, Tuple<TYPES...> >::get(t);
}
```

- Utilisation simplifiée
  - `get_value<1>(tuple)` ⇒ *2.7*

Parcours d'un pack d'arguments *(1/4)*

- Comment parcourir un pack d'arguments ?

- Solution 1: approche récursive
  - ```
    template <typename T>
    void display(const T & valeur) {
     cout << valeur << endl;
    }
    ```

  - ```
    template <typename T,typename... RESTE>
    void display(const T & v1,const RESTE &... reste) {
     cout << v1 << " ; ";
     display(reste...);
    }
    ```

  - `display(13,2.7,"Hello World !");` ⇒ *13 ; 2.7 ; Hello World !*

Code source complet: `cpp11_variadic_call.cpp`

Parcours d'un pack d'arguments *(2/4)*

- Solution 2: approche par expansion
  - ```
    template <typename T> void display(const T & valeur)
    { cout << valeur << " ; "; }
    ```
  - ```
    template <typename... TYPES>
    void display(const TYPES &... valeurs) {
     display(valeurs)...;
     cout << endl;
    }
    ```
    ⇒ expansion interdite ici !
  - Pourtant l'expansion semble valide
    - `display(valeurs)...` ⇒ `display(v1),` … `,display(vn)`
  - Mais l'expansion n'est autorisée qu'à certains endroits
    - Notamment ici: paramètre de *template*, argument de fonction, base d'héritage, liste d'initialisation d'un constructeur
    - `http://en.cppreference.com/w/cpp/language/parameter_pack`

Parcours d'un pack d'arguments *(3/4)*

- Solution 2 (suite)
  - On peut utiliser une fonction pour recevoir l'expansion

```
template <typename... TYPES>
inline void variadic_call(const TYPES &...) {}
```

  - 2ème tentative (presque bonne)

```
template <typename... TYPES>
void display(const TYPES &... valeurs) {
 variadic_call(display(valeurs)...);
}
```

  - Ne fonctionne que si «`display`» retourne une valeur

  - 3ème tentative (acceptable)

```
template <typename... TYPES>
void display(const TYPES &... valeurs) {
 variadic_call((display(valeurs),0)...);
}
```

Parcours d'un pack d'arguments *(4/4)*

- Solution 2 (suite et fin)
  - Mais l'ordre d'évaluation n'est pas garanti
    - `display(13,2.7,"Hello World !");` ⇒ *Hello World ! ; 2.7 ; 13 ;*
  - L'ordre peut être garanti par une liste d'initialisation
  - Version finale (ouf !)
    - ```
      struct variadic_call {
       variadic_call(const std::initializer_list<int>) {}
      };
      ```
    - ```
      template <typename... TYPES>
      void display(const TYPES &... valeurs) {
       variadic_call{(display(valeurs),0)...};
      }
      ```