Templates variadiques en C++

Johan Boulé

Variadique ?

- Une même fonction peut-être appelée avec un nombre d’arguments variable.
- La fonction est écrite sans connaître à-priori le nombre d’arguments : nombre d’arguments “dynamique” à chaque appel.
- L’appellant décide du nombre d’arguments qu’il transmet.
- Terme applicable à tout ce qui ressemble à une liste d’arguments, comme par exemple la liste des classes de base dans la définition d’un héritage, la liste des variables capturées par une expression lambda …

C-Style variadic functions

- Exemples “canoniques” : les fonctions printf et scanf
- Les arguments n’ont aucun type spécifié.
- Le nombre d’arguments doit être fourni par l’appellant (printf compte le nombre de caractère ‘%’ pour le déterminer)
- Ne fonctionne qu’avec les types dits “self-promoted”

Macros variadiques

- Simple expansion de texte
- Pas de mécanisme pour extraire une partie des arguments
- Pas de récursivité possible

Templates variadiques

- Sûreté du typage
- Vérifié statiquement lors de l'instanciation du template
- Une seule syntaxe : les points de suspension "…"
- Mécanisme de "dépaquetage" pour transmettre les arguments d'un template à un autre
- Possibilité de "dépaqueter" des expressions composées des arguments
- Un seul mécanisme d'extraction des arguments : appel "récursif" avec transmission du reste des arguments non traités.
- S'applique aussi bien aux types qu'aux valeurs => utile dans le domaine de la métaprogrammation pure
- Fonctionne aussi à des endroits inattendus :
  - dans les clauses d'héritage : la liste des classes de base dans la définition d'un héritage peut être variadique!
  - dans les clauses de capture des fonctions lambda,
  - dans les attributs "[[xxx]]" (non encore exploité).

A disposition dans les bilbliothèques

- Des classes et fonctions emblématiques du cas d'utilisation:
  - std::tuple
  - std::function
  - std::mem_fn
  - std::bind
  - std::invoke
  - std::integer_sequence …
- Et tout aussi intéressant : une utilisation pervasive dans de *nombreux* endroits :
  - std::make_share, make_unique,
  - std::thread::thread,
  - std::async,
  - placement new dans les allocateurs (std::allocator::construct)
  - emplace dans les conteneurs (std::vector)
  - std::result_of …

- Template metaprogramming :
  - Boost.Hana : refonte de Boost.MPL et Boost.Fusion en abandonnant C++03

- Des classes et fonctions standard C++03 sont rendues obsolètes depuis les templates variadiques :
  - std::unary_function, binary_function,
  - std::ptr_fun,
  - std::mem_fun,
  - std::bind1st, bind2nd …

Exemples (hors slides, voir fichiers annexes)

- Création d'une function variadique max de 3 façons :
  - avec le système de fonctions variadiques hérité du langage C => DANGER!,
  - avec les macros variadiques du préprocesseur,
  - et finalement, avec les templates variadiques du langage C++,
- Safe printf : un printf corrigé, avec sûreté du typage,
- Utilisation des fonctions optimisées « emplace » de la bibliothèque standard,
- Exemple de « perfect forwarding » pour l'héritage de constructeurs.

Vos réactions!