Isima

PARTIE VI

Généricité avancée

Bruno Bachelet

Loïc Yon

Nécessité des «concepts» *(1/3)*

- Exemple: algorithme de tri générique
  - Classe générique «`AlgoTri`» avec paramètre «`T`»
  - `T` = type des éléments à trier
  - Eléments comparés à l'aide de la méthode «`estAvant`»
    - Permet un tri décroissant par exemple

- Code possible

```
template <typename T>
void AlgoTri<T>::trier(T t[], int n) {
 for (int i = 0; i < n-1; i++)
  for (int j = i+1; j < n; j++)
   if (t[j].estAvant(t[i]))
   { T x = t[i]; t[i] = t[j]; t[j] = x; }
}
```

- Hypothèse: le type «`T`» possède la méthode «`estAvant`»
  - Vérification faite à la compilation, au moment de l'instanciation

Nécessité des «concepts» *(2/3)*

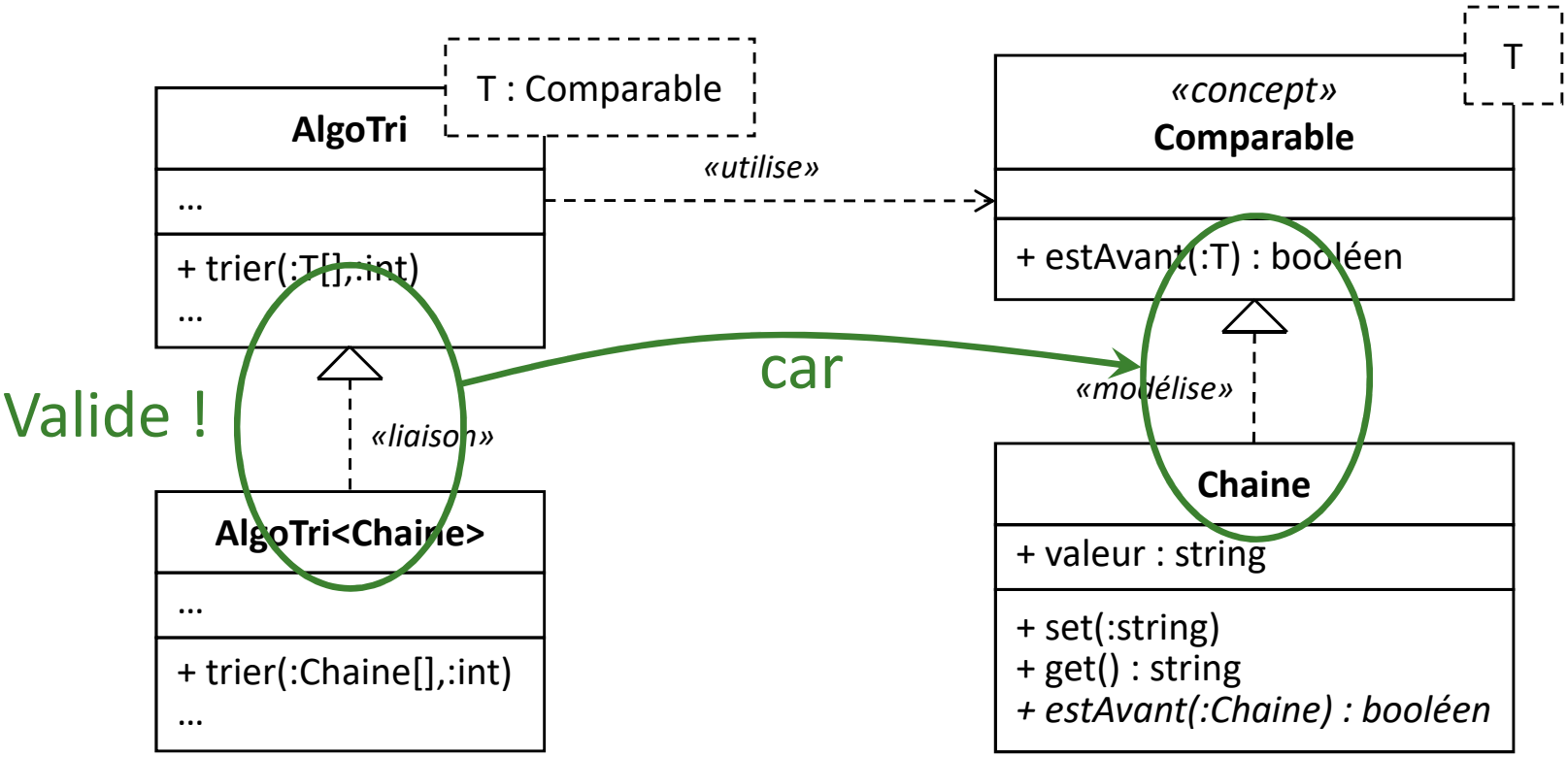
- L'interface supposée de «`T`» fait partie d'un «concept»

- Concept = ensemble «nommé» de spécifications/contraintes
  - Concerne l'interface: existence d'une méthode
  - Concerne l'implémentation: sémantique d'une méthode
  - Mais aussi toute contrainte pertinente liée à l'utilisation du type

- Proposition de définition du concept «`Comparable`»
  - La méthode d'instance «`estAvant`» doit exister
    - Signature exacte (UML): **booléen estAvant(:T)**
  - Elle doit définir une relation d'ordre (partielle) entre deux objets
    - Pseudo-code: **si a.estAvant(b) = vrai alors b.estAvant(a) = faux**

- Le type «`T`» de l'algorithme peut être contraint par ce concept
  - «`T`» respecte le concept «`Comparable`»
  - On dit aussi: «`T`» modélise le concept «`Comparable`»

Nécessité des «concepts» *(3/3)*



- En Java: concepts limités aux interfaces
  - Un paramètre générique peut être contraint par des interfaces
- En C++: concepts partiellement intégrés au langage depuis C++20
  - Avant: seule une documentation permettait de les identifier (cf. doc STL)
  - Maintenant: un paramètre générique peut être contraint par des concepts

Concepts et contraintes *(1/6)*

- Concept «`Comparable`»
  - ```
    template <typename T>
    concept Comparable = requires(const T a, const T b) {
     { a.estAvant(b) } -> std::convertible_to<bool>;
    };
    ```
  - «`a.estAvant(b)`» doit être valide pour les objets constants «`a`» et «`b`» de type «`T`»
    ⇒ la méthode «`estAvant`» doit exister et être constante
  - Et elle doit retourner une valeur convertible en booléen
    («`convertible_to`» est un concept standard)

- Pour formuler des exigences ⇒ expression «`requires`»
  - Syntaxe: `requires (`*paramètres optionnels*`) { `*exigences*` }`
  - Lister les exigences (*«requirements»*) à appliquer…
  - …aux paramètres du concept (le paramètre «`T`» dans l'exemple)
  - …aux paramètres locaux de l'expression (les variables «`a`» et «`b`» dans l'exemple)
  - Analysées à la compilation mais non évaluées à l'exécution
    ```

Concepts et contraintes *(2/6)*

- Contrainte sur la classe «`AlgoTri`» ⇒ mot-clé «`requires`»

```
template <typename T> requires Comparable<T>
class AlgoTri {
 public:
  void trier(T[],int);
};
```

- Syntaxe alternative

```
template <Comparable T> class AlgoTri {…}
```

- Possibilité aussi d'écrire la liste d'exigences directement

```
template <typename T> requires (
 requires(const T a, const T b) {
  { a.estAvant(b) } -> std::convertible_to<bool>;
 }
)
class AlgoTri {…}
```

Concepts et contraintes *(3/6)*

- Contrainte atomique
  - Modéliser un concept
    - `Comparable<T>` ⇒ «`T`» modélise «`Comparable`»
  - Valider une expression booléenne
    - `std::is_convertible<T,bool>::value` ⇒ «`T`» est convertible en booléen
    - «`is_convertible`» est un *«type trait»* standard
    - *Type trait* = métafonction qui vérifie une propriété d'un type (cf. métaprogrammation)
  - Liste d'exigences (expression «`requires`»)
    - `requires(T a, const T b) { a = b; }`

- Combinaison de contraintes
  - Conjonction
    - `Comparable<T> && Copiable<T>`
  - Disjonction
    - `std::integral<T> || std::floating_point<T>`

- Un concept est une contrainte nommée

Concepts et contraintes *(4/6)*

- 4 types de *«requirements»* pour formuler une liste d'exigences
- *Simple requirement*: impose qu'une expression est valide
  - `a + b;` ⇒ expression (non évaluée) valide (elle compile) ⇒ opérateur «+» existe
- *Type requirement*: impose qu'une expression est un type valide
  - `typename T::value_type;` ⇒ «T» possède le type interne «value_type»
  - `typename A<T>` ⇒ instanciation valide ⇒ «T» satisfait les contraintes de «A»
- *Compound requirement*: impose qu'une expression est valide
  + des contraintes sur son type de retour
  - `{ a * 2 } -> std::convertible_to<T>`
    ⇒ expression (non évaluée) valide et résultat convertible en «T»
- *Nested requirement*: impose des contraintes supplémentaires
  - `requires std::integral<T> || std::floating_point<T>`
    ⇒ «T» modélise l'un des deux concepts ⇒ «T» représente un entier ou un flottant

Concepts et contraintes *(5/6)*

- Concept à plusieurs paramètres ⇒ mécanisme de substitution
- Exemple: `std::convertible_to`
  - Concept standard à deux paramètres
  - `template <typename FROM, typename TO> concept convertible_to = …;`
- Dans une contrainte: type contraint ajouté implicitement comme premier paramètre du concept
  - `template <std::convertible_to<double> T>`
    ⇔ `requires std::convertible_to<T,double>`
  - `{ a * b } -> std::convertible_to<double>`
    ⇔ `requires std::convertible_to<decltype(a * b),double>`
  - Remarque: «`decltype`» détermine le type d'une expression sans l'évaluer (voir détails plus tard)

Concepts et contraintes *(6/6)*

- Exemple précédent (tri) incomplet

- «T» doit aussi être un type «copiable»
  - Opérateur d'affectation nécessaire
  - Constructeur de copie nécessaire

- Concept «Copiable»

```
template <typename T>
concept Copiable = requires(T a, const T b) {
 a = b;
 T(b);
};
```

- «T» est donc contraint par deux concepts

```
template <typename T>
requires (Comparable<T> && Copiable<T>) class AlgoTri {…};
```

Spécialisation statique

- Composant générique = modèle indépendant des types
- Mais cela peut être pénalisant
  - Exemple: recherche d'un élément dans une structure
  - Approches différentes suivant que la structure soit triée ou non

⇒ Mécanisme de spécialisation «statique»

  - Spécialisation du modèle générique pour un jeu de paramètres
  - Jeu de paramètres partiel ou complet
    - On parle aussi d'«instanciation» partielle ou complète

- Associé au polymorphisme statique de l'instanciation
  - «Meilleure» instanciation choisie en fonction du jeu de paramètres

Spécialisation d'une fonction générique

- Modèle générique d'une fonction de calcul de moyenne

```
template <int N> double moyenne(int * tab) {
 double somme = 0.0;
 for (int i = 0; i < N; ++i) somme += tab[i];
 return (somme/N);
}
```

- Spécialisation du modèle pour *N = 2* et *N = 1*

```
template <> double moyenne<2>(int * tab)
{ return (double(tab[0] + tab[1])/2); }

template <> double moyenne<1>(int * tab)
{ return double(tab[0]); }
```

- Attention
  - Déclarer d'abord la version générique, puis les versions spécifiques
  - Spécialisation «partielle» d'une fonction (ou méthode) interdite

Spécialisation d'une classe générique

- Modèle générique d'un vecteur d'éléments

```
template <typename T> class Vecteur {
 private:
  T * elements;
  int taille;
 …
 public: T operator[](int i) { return elements[i]; }
};
```

- Spécialisation du modèle pour *T = bool*

```
template <> class Vecteur<bool> {
 private:
  char * elements;
  int taille;
 …
 public: bool operator[](int i)
 { return ((elements[i/8] >> (i%8)) & 1); }
};
```

Polymorphisme statique

- Mécanisme statique lors de l'instanciation d'un modèle
  - Sélection de la version la plus spécialisée
  - En fonction du jeu de paramètres

⇒ Génération du code le plus dédié possible

- Exemples d'instanciation avec polymorphisme
  - Nombre entier (statique) inconnu: `moyenne<N>(tab);`
    - `N` = `10` ⇒ version générique
    - `N` = `2` ⇒ version spécialisée
  - Type inconnu: `Vecteur<T> v;`
    - `T` = `int` ⇒ version générique
    - `T` = `bool` ⇒ version spécialisée

- Concepts et contraintes peuvent servir à définir une spécialisation
  - `template <typename T> void f(T &);` ⇒ définition version générique
  - `template <std::integral T> void f(T &);` ⇒ définition version spécialisée

Spécialisation partielle *(1/2)*

- Spécialisation partielle
  = spécialisation avec un jeu de paramètres incomplet

- Retour sur l'exemple de calcul de moyenne

```
template <typename T, int N> class Moyenne {
 public: static T calculer(T * tab) {
  T somme = T();
  for (int i = 0; i < N; ++i) somme += tab[i];
  return (somme/T(N));
 }
};
```

- Spécialisation pour *N = 2* (*T* reste inconnu)

```
template <typename T> class Moyenne<T,2> {
 public: static T calculer(T * tab)
 { return ((tab[0] + tab[1])/T(2)); }
};
```

Spécialisation partielle *(2/2)*

- Instanciation partielle ⇒ passage par une classe
  - Rappel: instanciation partielle interdite pour une fonction
- Peut impliquer une lourdeur d'écriture
  - `double m = Moyenne<double,10>::calculer(tab);`
- Solution possible: proposer une fonction «aidante»

```
template <int N, typename T>
inline T moyenne(T * tab) {
 return Moyenne<T,N>::calculer(tab);
}
```

- Facilite l'instanciation grâce à la déduction automatique de type
  - `double m = moyenne<2>(tab);`
  - Remarque: l'ordre de déclaration des paramètres a son importance
    - «N» est fourni explicitement
    - «T» est déduit des arguments de la fonction

Retour sur l'héritage avec généricité

- Exemple
  - `template <typename T> class A`
    `{ … public: void m(); … };`
  - `template <typename T> class B : public A<T>`
    `{ … public: void n() { … m(); … } … };`

- Instanciation partielle ⇒ doute
  - La version générique de «`A`» peut être remise en question par une spécialisation
  - La méthode «`m`» peut ne pas exister dans cette spécialisation
    ⇒ appel à la fonction «`m`» si elle existe au lieu de la méthode

- Conseil: toujours utiliser «`this->`» sur un membre hérité
  - Cela provoquera une erreur si une spécialisation ne possède pas ce membre
  - Toujours mieux qu'un comportement implicite (très probablement non voulu)
  - Solution: `void n() { … this->m(); … }`

Alias de type

- Deux syntaxes possibles
  - C++03: `typedef pair<int,double> paire_t;`
  - C++11: `using paire_t = pair<int,double>;`
  - Strictement équivalentes ⇒ privilégier la seconde
- Alias de type *template* possible avec «`using`»
  - `template <typename T> using paire_t = pair<int,T>;`
  - Forme d'instanciation «partielle»
- Type interne à une classe
  - Possibilité de déclarer une classe dans une autre
    - `class vector { … class iterator { … }; … };`
  - Et de déclarer des alias de type
    - `class vector { … using iterator = …; … };`

Mot-clé «*typename*»

- Indique que ce qui suit est un type
- Utilisé dans la déclaration d'un paramètre
  - `template <typename T>`
  - Equivalent à: `template <class T>`
- Utilisé pour lever une ambiguïté
  - A cause de l'instanciation partielle
  - Tant que l'instanciation n'est pas effective ⇒ doute
  - Exemple
    - ```
      template <typename T> class vector
      { public: using value_type = T; /* Type interne */ };
      ```
    - ```
      template <typename T> class B
      { public: using type_elt = typename vector<T>::value_type; };
      ```
  - Doute sur la nature de «`vector<T>::value_type`»
    - Type ou attribut ? ⇒ `typename`

Inférence de type *(1/3)*

- Déclarer le type d'une variable n'est pas toujours évident
  - Compliqué à écrire (e.g. instanciation d'un *template*)
  - Polymorphisme statique ⇒ difficile de connaître le type d'un retour
- Alors que le compilateur peut le déduire
  - Contrôle des types à la compilation
  - Capable de détecter une erreur de type ⇒ capable de corriger
- Exemple

```
std::vector<int> v = {…};
? it = std::find(v.begin(),v.end(),5);
if (it != v.end()) *it = 0;
```

Inférence de type *(2/3)*

- Première possibilité: mot-clé «`auto`»
  - Attention au changement de signification
  - C++03: `auto int x;` ⇒ incorrect en C++11
  - C++11: `auto x = …;` ⇒ déduction du type

- `auto` = joker
  - Le programmeur laisse le compilateur déduire
  - `auto it = std::find(v.begin(),v.end(),5);`

- Peut remplacer le retour d'une fonction
  - `auto add(double a, double b) { return a + b; }`
  - ```
    template <typename T, typename U>
    auto add(const T & a, const U & b) { return a + b; }
    ```

- Peut remplacer tout ou partie d'un type
  - `auto * x, auto & x, const auto & x…`
  - `auto x = new auto(5);`
    ```

Inférence de type *(3/3)*

- Seconde possibilité: `decltype(`*expression*`)`

- Identifie le type d'une expression
  - Demande au compilateur de déduire le type
  - Mais l'expression n'est pas compilée, ni évaluée à l'exécution

- Exemple

```
decltype(v.begin()) it;
it = std::find(v.begin(),v.end(),5);
```

- Quelle que soit l'approche, le type est connu à la compilation
  - Les contrôles de types sont donc préservés

- Mais le type n'est pas explicite dans le code

Variadic template (1/2)

- *Variadic template* = générique à paramètres variables
  - Depuis C++11
  - 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;`

- Syntaxe simple pour instancier le générique
  - `Tuple<int,double,std::string> t;`

- Mais syntaxe pas toujours intuitive pour écrire le générique
  - Mécanisme d'«expansion»
  - Ou approche récursive (métaprogrammation)

Variadic template (2/2)

- Paramètres variables = «pack» de paramètres
  - Pack représenté par le symbole «`...`»
  - Pack = 0 à *n* paramètres
- Pack de types: `template <typename... TYPES>`
- Pack de nombres: `template <int... VALEURS>`
- Pack d'arguments: déclaration à partir d'un pack de types
  - `TYPES... args` ⇒ *n* arguments, chacun d'un type du pack
- Permet de renforcer le contrôle de types des arguments variables
  - `fprintf(const char * format, ...);`
    - Impossible d'identifier les types des arguments variables
  - ```
    template <typename... TYPES>
    void fprintf(const char * format, TYPES... args);
    ```
    - Possibilité d'identifier le type de chaque argument
    ```

Accès aux éléments d'un pack *(1/2)*

- `template <typename... TYPES>`
  ⇒ `template <typename` T_1`, …, typename` T_n`>`
  - Il s'agit d'une illustration: $T_1 \ldots T_n$ n'existent pas explicitement

- Accès à un paramètre d'un pack impossible directement
  - 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

- Nombre d'éléments d'un pack: opérateur `sizeof...(`*PACK*`)`

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

Accès aux éléments d'un pack *(2/2)*

- Exemple: implémentation possible de «`sizeof...`»
  - Approche récursive par spécialisation de *template*
  - Utilisation du *template*: `taille<`*PACK*`>::val`
- Version primaire = déclaration
  - `template <typename... TYPES> struct taille;`
- Spécialisation n°1 = récursion
  - *taille({ T_1, …, T_n }) = taille({ T_2, …, T_n }) + 1*
  - ```
    template <typename PREMIER, typename... RESTE>
    struct taille<PREMIER,RESTE...>
    { static const int val = taille<RESTE...>::val + 1; };
    ```
- Spécialisation n°2 = arrêt
  - *taille({ }) = 0*
  - ```
    template <> struct taille<>
    { static const int val = 0; };
    ```

Expansion de pack *(1/2)*

- Autre possibilité d'utilisation d'un pack ⇒ mécanisme d'expansion
- On décrit un schéma d'expansion
  - Expression contenant l'identifiant d'un pack
  - Et terminée par « `...` »
  - Exemples
    - `TYPES...`
    - `const TYPES &...`
    - `vector<TYPES>...`
- Expansion ⇒ réplication du schéma
  - Pour chaque paramètre du pack
  - Séparation par une virgule
  - Exemples
    - `TYPES...` ⇒ T_1, …, T_n
    - `vector<TYPES>...` ⇒ `vector<`T_1`>`, …, `vector<`T_n`>`
    - `const TYPES &... x` ⇒ `const` T_1 `&` x_1, …, `const` T_n `&` x_n

Expansion de pack *(2/2)*

- Le mécanisme d'expansion s'applique aussi au pack d'arguments
  - ```
    template <typename... TYPES>
    void f(TYPES... args) { g(args...); }
    ```
- La localisation de « `...` » est importante
  - Exercice: trouver les expansions suivantes (réf. Andrei Alexandrescu)
    - `g(A<TYPES>::m(args)...);`
    - `g(A<TYPES...>::m(args...));`
    - `g(A<TYPES...>::m(args)...);`
- Depuis C++17: introduction des *«fold expressions»*
  - Expansion possible avec les opérateurs binaires
  - Exemples
    - `(args + ...)` ⇒ $(arg_1 + (\ldots + (arg_{n-1} + arg_n)))$
    - `(... + args)` ⇒ $(((arg_1 + arg_2) + \ldots) + arg_n)$
    ```