Module R3.04

Conception et Programmation Objet Avancées

Cours 2 – Généricité : *templates*, STL

IUT2
Université Grenoble Alpes
Département
INFO

Module R3.04
Chapitre 5

Généricité : *templates*

Généralités

- Les ***templates*** permettent d'écrire des fonctions et des classes **génériques**.
- Un *template* est morceau de code que l'on va écrire en manipulant un ou des types qui ne sont pas encore connus.
  Ces types inconnus sont appelés ***paramètres template***
- Lorsque l'on va utiliser le *template*, il va falloir donner une valeur (des types existants !) à ces ***paramètres template***.
  On parle d'**instancier le template.**
- Un paramètre **template** peut être soit :
  - un **type (générique)** : on parle de **paramètre de type**
  - un paramètre « ordinaire » : on parle de **paramètre expression**
- Les fonctions et les classes ainsi paramétrées sont appelées respectivement **patron de fonction** et **patron de classe**
- On doit utiliser un ***template*** dès que l'on s'aperçoit que l'on a besoin de dupliquer plusieurs fois le même code en ne changeant que les types manipulés dans ce code

Patron de fonction

- Un **patron de *fonction*** est donc une fonction générique :
  - dont certains paramètres sont de **types génériques**. Un type générique est appelé **paramètre de type** du patron de fonction.
  - dont les éventuels autres paramètres sont des paramètres « ordinaires ». Ces paramètres sont des **paramètres expression** du patron de fonction
- Chaque **paramètre de type** doit apparaître au moins une fois dans l'en-tête du patron
- Exemple : patron "minimum de 2 valeurs a et b de type **T**"

```
template <class T>
```

Paramètres du template

```
T min (T a, T b) {
  if (a<b) return a; else return b;
}
```

Fonction générique ***min<T>***

  - **T** est un "paramètre de type" du patron de fonction **`min<T>`**
  - a et b sont des "paramètres expression" du patron **`min<T>`**

Patron de classe

- Un **patron de classe** est une classe générique qui fait intervenir, dans la définition de ses membres (attributs ou méthodes) :
  - Des types génériques : un tel type est donc considéré comme un **paramètre de type** du patron de classe
  - Des **paramètres expressions** : ce sont des paramètres « ordinaires » (int, float, ..) qui permettent de paramétrer la définition du patron.
- Exemple : patron "tableau de n éléments de type T"

```
template <class T, int n=10>
class Tableau {
    T m_tab[n];
  public:
    Tableau();
    void ajouterElement(T e);
    ...
};
```

  - T est un paramètre de type du patron de classe **Tableau<T>**
  - **n** est un paramètre expression du patron de classe **Tableau<T>**
  - on peut donner une valeur par défaut aux paramètres "*templates*"

Généralités (suite)

- La génération du code (l'*instanciation des patrons*)
  - les paramètres génériques sont remplacés par de vrais types *(c'est un vrai « rechercher/remplacer » !)*
  - les paramètres expressions prennent leur valeur
- L'instanciation du patron a lieu lorsque l'on utilise le patron de fonction ou de classe pour la 1ère fois.
- Les types réels à utiliser à la place des types génériques sont déterminés par le compilateur lors de cette première utilisation :
  - *Implicitement à partir du contexte d'utilisation (déprécié)*
  - *Explicitement* par les paramètres donnés lors de l'utilisation du patron

C++ permet donc définir :

- Des fonctions et des classes **génériques**
  appelées respectivement patrons de fonction et patrons de classe

- Les **paramètres de généricité** sont essentiellement des **types de données**
  - types scalaires de base (`int`, `float`, ...)
  - Classes

- C++ n'est pas le seul langage à « offrir » ce mécanisme de généricité : vous l'avez utilisé en **Ada** et en **Java**

Exemple de procédure générique

- Définition d'une procédure patron permettant de permuter le contenu de 2 variables d'un type quelconque (fichier `permut.h`)

```
// paramètre de généricité
// T représente le type des variables permutées
// Les caractères <> sont obligatoires

template <class T>
void permut(T & e1, T & e2) {

    T e = e1;
    e1 = e2;
    e2 = e;
}
```

Exemple d'utilisation (instanciation)

- Utilisation pour permuter des Personnes, des entiers des float

```
#include "Personne.h"
#include "permut.h"
int main() {

  Personne p1("dupond", "10 rue des fleurs");
  Personne p2("durand", "15 rue des cactus");
  // permut<Personne>(Personne&, Personne&)
  permut<Personne>(p1, p2); // 1ère Instanciation

  int i1=3, i2=5;
  // permut<int>(int&, int&)
  permut<int>(i1, i2);       // 2ème Instanciation

  float f1=2.0, f2=5.0;
  // permut<float>(float&, float&)
  permut<float>(f1,f2);      // 3ème Instanciation

  return 0;
}
```

Classe Tableau générique : Tableau.h

```
// paramètre de généricité T = type des éléments du tableau

template <class T>
class Tableau {

protected:
  int   m_tailleMax;  // Nombre max d'éléments de m_tab
  T *   m_tab;        // Tableau dynamique d'éléments de type T
  short m_taille;     // Nombres d'éléments présents dans m_tab

public:
        Tableau   (int tailleMax);  // Construit un tableau vide
  int   getTaille ();               // Retourne la taille du tableau
  void  ajouter   (T e);            // Ajoute e au tableau
  void  supprimer (int i);          // Supprime le ième élém.
  T &   operator[](int i);          // Retourne l'élém. i s'il existe

};
```

// Template : Il faut implémenter les méthodes ici, dans le fichier .h

Implémentation : Tableau.h (suite)

```
// Il faut implémenter les méthodes ICI, dans le fichier .h

template <class T>
Tableau<T>::Tableau(int tailleMax) {
  : m_taille(0),
    m_tailleMax(tailleMax)
{
    m_tab = new T [m_tailleMax];
}

template <class T>
int Tableau<T>::getTaille() {
  return m_taille;
}
```

Implémentation : Tableau.h (suite)

```
template <class T>
void Tableau<T>::ajouter(T e) {
  if ( m_taille < m_tailleMax ) m_tab[m_taille++]=e;
  else { throw "Opération Impossible"; // levée d'exception }
}

template <class T>
void Tableau<T>::supprimer(short i) {
  if (i>=0 && i < m_taille ) {
    for (int j=i; j<m_taille-1; j++) m_tab[j] = m_tab[j+1];
    m_taille--;
  }
}

template <class T>
T & Tableau<T>::operator[](short i) {
  if ( i<0 || i >= m_taille ) { throw "Accès Interdit"; }
  return m_tab[i];
}
```

Exemple d'instanciations

```
#include <iostream>
#include "Personne.h"
#include "Tableau.h"

int main() {
  // Instanciation 1 : tableau de 10 entiers
  Tableau <int> t1(10);
  t1.ajouter(1); t1.ajouter(2); t1.ajouter(3);
  for (int i=0; i<t1.getTaille(); i++)
    std::cout << t1[i] << endl;

 // Instanciation 2 : tableau de 10 pointeurs sur des personnes
  Tableau <Personne *> t2(10);
  t2.ajouter(new Personne("dupond", "10 rue des fleurs"));
  t2.ajouter(new Personne("durand", "25 rue des fleurs"));
  for (int i=0; i<t2.getTaille(); i++)
    t2[i]->afficher();

  return 0;
}
```

Remarque 1

- Chaque méthode est précédée de la déclaration suivante qui rappelle les paramètres de généricité :

```
template <class T>
```

- Le nom de chaque méthode est précédé du nom de la classe qui est paramétré, entre chevrons (<>), par le paramètre de généricité

```
void Tableau<T>::supprimer(short i)
```

Remarque 2

- Dans un programme qui utilise un patron, **il faut inclure l'implémentation de ce template.** Deux solutions :
  - **Solution 1 (conseillé)** : on écrit la spécification et l'implémentation dans un seul fichier : **`MonTemplate.h`**
  - ***Solution 2 :*** *on peut aussi écrire la spécification et l'implémentation dans deux fichiers distincts,* **`MonTemplate.h`** *et* **`MonTemplate.cpp`**, *mais l'implémentation devra alors être incluse dans la spécification (et non l'inverse) :*

MonTemplate.h

```
#ifndef MONTEMPLATE_H
#define MONTEMPLATE_H
// spécification template
#include "MonTemplate.cpp"
#endif
```

MonTemplate.cpp

```
#ifdef MONTEMPLATE_H
// on n'inclut pas le .h
// implémentation template
#endif
```

- Dans les deux cas, on inclura "MonTemplate.h" lorsque l'on voudra utiliser (instancier) le *template*.

Module R3.04
Chapitre 6

Standard Template Library (STL)

Présentation très rapide

La librairie C++ standard

- C'est une collection de plusieurs éléments (fonctions, constantes, classes, objets et patrons (*templates*)) qui étendent le langage C++
- Cette librairie fournit des fonctionnalités de base :
  - Chaînes de caractères : **string Library**
  - Entrées/Sorties via des flux : **Input / Output Streams Library**
  - Compatibilité avec le C : **C Library**
  - Conteneurs de données : **Standard Template Library (STL)**
- Les déclarations des différents éléments de la librairie C++ standard sont réparties dans plusieurs fichiers qui doivent être inclus dans votre code pour avoir accès aux fonctionnalités dont vous avez besoin :

```
#include <string>
#include <vector>
…
```

- Les éléments proposés par la librairie standard C++ sont définis dans l'espace de nom standard (**`std`**). Pour les utiliser il faut donc :
  - soit faire précéder chaque identificateur par **`std::`**
    `std::string, std::vector, std::list, std::sort, …`
  - soit ajouter une instruction **`using namespace std`**

Standard Template Library (STL)

- Cette librairie définit plusieurs sortes de **conteneurs** de données
- Un **conteneur** est un objet permettant de stocker/manipuler une collection de données
- Le type des données stockées dans un conteneur peut être quelconque : les conteneurs sont donc proposés par la STL sous forme de **templates** (patrons). **Mais attention : on ne peut pas stocker de références (&) dans un conteneur de la STL**
- La STL offre un mécanisme unifié pour accéder aux éléments d'un conteneur, quel que soit le type de ce conteneur : c'est la notion **d'itérateur** (très proche du pointeur)
- Grâce à cette notion d'itérateur, la STL propose également, sous forme de fonctions, une collection **d'algorithmes** standards qui peuvent s'appliquer au contenu (ou à une partie du contenu) de n'importe quel conteneur.

Les patrons "conteneurs" de la STL (1)

- **1. Les conteneurs séquentiels**
  Pour stocker une séquence (!) d'éléments

  - **vector<T>** : le vecteur traditionnel, mais dynamique (taille variable)
    - Collection d'éléments indicés (à partir de 0), stockés de manière
    - Optimisé pour des accès aléatoires (v[i]) et pour faire des ajouts/suppression en fin de collection
  - **deque<T>** (**d**ouble-**e**nded **queue**) :
    - Interface proche du vecteur (éléments indicés)
    - Optimisé pour insertion/suppression en début ou en fin
  - **list<T>** : la liste doublement chaînée traditionnelle
    - Collection d'éléments stockés de manière **non** contigüe en mémoire
    - Optimisé pour ajout/suppression n'importe où et parcours bi-directionnel
  - **array<T>** (C++11) : conteneur de taille fixe, contenu ordonné
  - **forward_list<T>** (C++11) : liste simplement chaînée

Les patrons "conteneurs" de la STL (2)

- **2. Les conteneurs "adaptateurs"**

  Adaptateurs car ces conteneurs encapsulent un autre conteneur dont ils sont, en quelque sorte, une adaptation

  - **stack<T>** : pile LIFO (*Last In First Out*)
  - **queue<T>** : queue FIFO (*First In First Out*)
  - **priority_queue<T>** : le premier élément est toujours le plus grand de la collection, selon un critère à définir

Les patrons "conteneurs" de la STL (3)

- **3. Les conteneurs associatifs :**
  Pour stocker des éléments de la forme (clé, valeur associée), ou simplement (clé), et accéder facilement à un élément grâce à la valeur de sa clé.

  - **set<T>** : pour stocker une collection d'éléments uniques et ordonnés (deux éléments ne peuvent pas avoir la même valeur)
  - **multiset<T>** : identique à set mais permet aussi de stocker des éléments ayant la même valeur
  - **map<keyT, valueT>** : mémoire associative : collection d'éléments qui sont des couples (clé, valeur), ordonnés selon les clés. Chaque élément à une clé unique. On peut accéder rapidement à un élément en fournissant sa clé.
  - **multimap <keyT, valueT>** : identique à **map** mais permet que plusieurs éléments aient la même clé

Les patrons "conteneurs" de la STL (4)

- **4. Les conteneurs associatifs non ordonnés (C++11)**

Identiques aux conteneurs associatif mais leur contenu n'est pas ordonné

- **unordered_set<T>**
- **unordered_ multiset<T>**
- **unordered_ map<keyT, valueT>**
- **unordered_ multimap <keyT, valueT>**

Les algorithmes de la STL

- C'est un ensemble de fonctions qui implémentent des algorithmes "standards" :
  - recherche, copie, tri, suppression, fusion, permutations, partitionnement,… etc.
- Ces fonctions opèrent en général sur une (ou des) partie(s) quelconque(s) de conteneur(s) définie(s) grâce à deux itérateurs :
  - un itérateur (first) pointant sur le premier élément de la partie à traiter
  - un autre itérateur (last) pointant sur le dernier élément de la partie à traiter

Les itérateurs

- **Iterateur** : objet associé à un conteneur, qui permet de "balayer" l'ensemble des objets du conteneur, sans connaître la structure de données sous-jacente
- Unifie la syntaxe et les algorithmes (tous les conteneurs peuvent se parcourir de la même façon)
- **Un itérateur s'utilise comme un pointeur** :
  - L'itérateur "pointe" sur un élément du conteneur
  - L'opérateur de déréférencement `*` s'applique à un itérateur pour obtenir l'élément pointé par cet itérateur
  - L'opérateur `++` permet de faire avancer un itérateur vers l'élément suivant dans le conteneur
- On peut distinguer différents types d'itérateurs :
  - RandomAccessIterator, BidirectionalIterator, ForwardIterator, InputIterator, OutputIterator
  - Selon le type de l'itérateur, d'autres opérateurs que `*` et `++` peuvent être appliqués (`--`, `+`, `-`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `operator []`, `+=`, `-=`)

Exemple : le template Vector (1)

- Implémente un tableau traditionnel (éléments indicés, stockés consécutivement en mémoire)
  La taille du vecteur peut varier dynamiquement
- Méthodes de base :
  - **`v.size()`** : Retourne le nombre d'éléments contenus dans **`v`**
  - **`v[i]`** : Retourne l'élément d'indice **`i`**
  - **`v.push_back(e)`** : Ajoute un élément de valeur **`e`** à la fin du vecteur
  - **`v.pop_back()`** : Supprime le dernier élément
  - **`v.begin()`** : Retourne un itérateur sur le premier élément
  - **`v.end()`** : Retourne un itérateur juste après le dernier élément
  - **`v.insert(it,e)`** : Insère un élément de valeur **`e`** DEVANT l'élément pointé par l'itérateur **`it`**.
    Après l'insertion, **`it`** pointe sur l'élément inséré

http://www.cplusplus.com/reference/vector/vector/

Exemple : le template vector (2)

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
  vector<float> v;                          // v={}, v.size()=0
  // Ajout d'éléments à la fin du vecteur
  v.push_back(10.5); v.push_back(20.3); // v={10.5, 20.3}, v.size()=2

  // Parcours en utilisant des indices
  for (unsigned int i = 0; i < v.size(); i++) cout << v[i] << endl;

  // Parcours en utilisant des itérateurs
  vector<float>::iterator it;          // déclaration d'un itérateur sur un vecteur
  for (it = v.begin(); it < v.end(); it++) cout << *it << endl;

  // Insertion d'un élément, en position 1 par exemple
  it = v.insert(v.begin() + 1, 8.2); // v={10.5, 8.2, 20.3}, v.size()=3
                                     // it pointe sur v[1] après l'insert

  // Suppression du dernier élément puis tri
  v.pop_back();                      // v={10.5, 8.2}, v.size()=2
  sort(v.begin(), v.end());          // v={8.2, 10.5}, v.size()=2

  // Sucres syntaxiques : parcours en utilisant "auto" et le "for" de la STL (c++011)
  for (auto & f : v) cout << f << endl; // equivalent à for(float & f : v)…
  return 0;
}
```

Exemple : le template list (1)

- Implémente un liste doublement chaînée (chaque élément peut être stocké n'importe où en mémoire, il possède un pointeur vers l'élément précédent dans la liste et un autre pointeur vers l'élément suivant).

- Méthodes de base :
  - **`l.push_front(e)`** : Ajoute un élément e en début de liste **`l`**
  - **`l.push_back(e)`** : Ajoute un élément e en fin de liste **`l`**
  - **`l.pop_front()`** : Supprime le premier élément de la liste **`l`**
  - **`l.pop_back()`** : Supprime le dernier élément de la liste **`l`**
  - **`l.sort()`** : Trie la liste **`l`**
  - **`l1.merge(l2)`** : fusion des listes **`l1`** et **`l2`** qui doivent être triées.
    Tous les éléments de **`l2`** sont insérés dans **`l1`**
    A la fin, **`l1`** est triée et **`l2`** est vide
  - **`l.unique()`** : Supprime, dans chaque groupe d'éléments égaux consécutifs, tous les éléments, sauf le premier.
    Donc, si **`l`** est triée, supprime les doublons.
  - **`l1.swap(l2)`** : Permute le contenu des listes **`l1`** et **`l2`**
  - **`l.reverse()`** : Inverse l'ordre des éléments de la liste **`l`**
  - **`l.remove(e)`** : Supprime de la liste **`l`** tous les éléments égaux à **`e`**

http://www.cplusplus.com/reference/list/list/

Exemple : le template list (2)

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main() {
  // Déclaration d'une liste de chaînes
  list<string> l;         // l est vide (l.size()=0)

  // Insertion d'éléments en queue de liste
  l.push_back("zebre");  // l = {"zebre"}
  l.push_back("auroch"); // l = {"zebre", "auroch"}

  // Insertion d'éléments en tête de liste
  l.push_front("gnu");   // l = {"gnu", "zebre", "auroch"}

  // tri de la liste
  l.sort();              // l = {"auroch", "gnu", "zebre"}

  // Parcours de la liste
  for (auto & chaine : l )  cout << chaine  << endl;

  return 0;
}
```

Avantages de la STL

- Ce sont les avantages de la **RE-U-TI-LI-SA-TION** :
  - On ne réinvente pas inutilement la roue
  - On gagne du temps de développement
  - On évite les bugs (la STL est bien programmée !)
  - On produit un code plus lisible (tous les développeurs C++ connaissent la STL)
  - On utilise des bibliothèques bien documentées **`http://www.cplusplus.com/reference/stl/`**
  - On travaille à un niveau d'abstraction supérieur (le concept de vecteur ou de liste chaînée devient une *brique de base* du travail de développement)
  - *…etc*

- **Conclusion** :
  - En C++, dès que possible, il faut utiliser les outils proposés par la librairie standard en général, et par la STL en particulier !
  - Développer dans un langage de haut niveau (C++, Java, C#, PHP7, …) qui propose de telles bibliothèques et ne pas les utiliser relève de la **FAUTE PROFESSIONNELLE** !