C++ Avancé (2° Partie)

Bjarne Stroustrup a développé C++ au cours des années 1980, alors qu'il travaillait dans le laboratoire de recherche Bell d'AT&T.

Le langage C++ est normalisé par l'ISO. Sa première normalisation date de 1998 (ISO/CEI 14882:1998), sa dernière de 2003 (ISO/CEI 14882:2003).

La composition et l'héritage fournissent un moyen de réutiliser le code objet.

La fonctionnalité template fournit, en C++, un moyen de réutiliser du code source.

Table des matières

Les Templates

- Les templates permettent d'écrire des fonctions et des classes en paramétrant le type de certains de leurs constituants (type des paramètres ou type de retour pour une fonction, type des éléments pour une classe collection par exemple).
- Les templates permettent d'écrire du code générique, c'est-à-dire qui peut servir pour une famille de fonctions ou de classes qui ne diffèrent que par la valeur de ces paramètres.
- En résumé, l'utilisation des templates permet de « paramétrer » le type des données manipulées.

➢ *Remarque : dans la bibliothèque standard C++, on trouve de nombreux templates (par exemple, les entrées/sorties, les chaînes de caractères ou les conteneurs).*

Besoins et Avantages

- Il faut parfois écrire de nombreuses versions d'une même fonction ou classe suivant les types de données manipulées.
- Par exemple, un tableau de `int` ou un tableau de `double` sont très semblables, et les fonctions de tri ou de recherche dans ces tableaux sont identiques, la seule différence étant le type des données manipulées.

- Avantages à utiliser des templates :
  - écritures uniques pour les fonctions et les classes.
  - moins d'erreurs dues à la réécriture.
  - performances améliorées grâce à la spécialisation en fonction des types de données.

Déclaration d'un modèle

- `template <class|typename nom[=type] [, class|typename nom[=type][...]>`
  - où nom est le nom que l'on donne au type générique dans cette déclaration. Le mot clé `class` a ici exactement la signification de "type". Il peut d'ailleurs être remplacé indifféremment dans cette syntaxe par le mot clé `typename`. On peut déclarer un nombre arbitraire de types génériques, en les séparant par des virgules.

- `template <type paramètre[=valeur][, ...]>`
  - où type est le type du paramètre constant, paramètre est le le nom du paramètre et valeur est sa valeur par défaut.

Fonctions et Classes Template

- Après la déclaration d'un ou de plusieurs paramètres "template" suit en général la déclaration ou la définition d'une fonction ou d'une classe "template".
- Dans cette définition, les types génériques peuvent être utilisés exactement comme s'il s'agissait de types normaux. Les constantes "template" peuvent être utilisées dans la fonction ou la classe "template" comme des constantes locales.

- Fonction template : `template<class T> T mini(T a, T b);`
- Classe template : `template<class T> class Array { T a ; };`

➢ *Remarque : Le template de fonction est généralement appelé algorithme (comme la plupart des templates de fonctions dans la bibliothèque standard du C++). Il dit juste comment faire quelque chose.*

Exemple de Fonction Template


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

template<class T> T mini(T a, T b)
{     if(a < b)   return a;
      else        return b;
}

template<class T> T mini(T a, T b, T c)
{     return mini(mini(a, b), c); }

int main()
{
   int n=12, p=15, q=2;
   float x=3.5, y=4.25, z=0.25;

   cout << "mini(n, p) -> " << mini(n, p) << endl; // implicite
   cout << "mini(n, p, q) -> " << mini(n, p, q) << endl;
   cout << "mini(x, y) -> " << mini(x, y) << endl;
   cout << "mini(x, y, z) -> " << mini(x, y, z) << endl;
   cout << "mini(n, p) -> " << mini<float>(n, q) << endl; // explicite
}
```

Exemple de Classe Template


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

template<class T> class Array {
      enum { size = 100 };
      T A[size];
   public:
      T& operator[](int index) {
         if(index >= 0 && index < size)   return A[index];
         else cerr << "Index out of range" << endl;
      }
};

int main() {
   Array<int> ia; Array<float> fa;

   for(int i = 0; i < 20; i++) {
      ia[i] = i * i; fa[i] = float(i) * 1.414; }
   for(int j = 0; j < 20; j++) cout << j << ": " << ia[j]
                                    << ", " << fa[j] << endl;
}
```

Spécialisation

- Lorsqu'une fonction ou une classe "template" a été définie, il est possible de la spécialiser pour un certain jeu de paramètres "template". Il faut faire précéder la définition de cette fonction ou de cette classe par la ligne suivante :

```
template <>
```

- qui permet de signaler que la liste des paramètres "template" pour cette spécialisation est vide (et donc que la spécialisation est totale).
- Si un seul paramètre est spécialisé, on parle de spécialisation partielle.

Exemple de spécialisation


```
template <> const char *mini(const char *a, const char *b)
{
    return (strcmp( a, b ) < 0) ? a : b;
}

cout << mini("hello", "wordl") << endl;
```

Définitions de méthodes non inline

- Dans ce cas, le compilateur a besoin de voir la déclaration template avant la définition de la fonction membre.

```
template<class T> class Array {
      enum { size = 100 };
      T A[size];
   public:
      T& operator[](int index);
};

template<class T> T& Array<T>::operator[](int index)
{  if(index >= 0 && index < size)   return A[index];
   else cerr << "Index out of range" << endl;
}

int main() {
   Array<float> fa; fa[0] = 1.414; }
```

Fichiers d'en-tête

- Même si vous créez des définitions de fonctions non inline, vous voudrez généralement mettre toutes les déclarations et les définitions d'un template dans un fichier d'en-tête.

- Ceci paraît violer la règle normale des fichiers d'en-tête qui est "Ne mettez dedans rien qui alloue de l'espace de stockage", (ce qui évite les erreurs de définitions multiples à l'édition de liens), mais les définitions de template sont spéciales.

- Tout ce qui est précédé par template<...> signifie que le compilateur n'allouera pas d'espace de stockage pour cela à ce moment, mais, à la place, attendra qu'il lui soit dit de le faire (par l'instanciation du template).

Inconvénients

- Cela signifie que les fichiers d'en-tête doivent contenir non seulement la **déclaration**, mais également la **définition** complète des "template".

- Cela a plusieurs inconvénients :
  - difficile de séparer la déclaration de la définition dans des fichiers séparés
  - les instances des "template" sont compilées plusieurs fois (diminue les performances des compilateurs)
  - les instances des "template" sont en multiples exemplaires dans les fichiers objets générés par le compilateur, et accroissent donc la taille des fichiers exécutables à l'issue de l'édition de liens.

Solutions

- Réduction du nombre de fichiers sources ou exporter les définitions des "template" dans des fichiers complémentaires.
- Utilisation des options des compilateurs. Actuellement, la plupart des compilateurs sont capables de générer des fichiers d'en-tête précompilés, qui contiennent le résultat de l'analyse des fichiers d'en-tête déjà lus.
- modification de l'éditeur de liens pour qu'il regroupe les différentes instances des mêmes "template".
- Enfin, certains compilateurs permettent de désactiver les instanciations implicites des "template". Cela permet de laisser au programmeur la responsabilité de les instancier manuellement, à l'aide d'instanciations explicites. Ainsi, les "template" peuvent n'être définies que dans un seul fichier source.

Une Solution Possible

templateArray.cc

```
#include "templateArray.h"
```

Extrait du man g++ :

C++ source files conventionally use one of the suffixes .C, .cc, .cpp, .CPP, .c++, .cp, or .cxx; C++ header files often use .hh, .hpp, .H, or (for shared template code) .tcc

templateArray.h

```
#ifndef _TEMPLATEARRAY_H_
#define _TEMPLATEARRAY_H_

#include <iostream>
using namespace std;

template<class T> class Array {
      enum { size = 100 };
      T A[size];
   public:
      T& operator[](int index);
};

#include "templateArray.tcc"

#endif
```

templateArray.tcc

```
template<class T> T& Array<T>::operator[](int index)
{  if(index >= 0 && index < size) return A[index];
   else cerr << "Index out of range" << endl;
}
```

Bibliographie

- Cours d'Éric REMY : http://pluton.up.univ-mrs.fr/eremy/Ens/Info1.C++/CM/C++.html

- Cours et tutoriels C++ : http://cpp.developpez.com/cours/