Table des matières

Besoin

En C++, la fonctionnalité *template* fournit un **moyen de réutiliser du code source**.

Objectifs

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.

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

Fonction et classe 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.

- 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 n°1 : Fonction template

On désire écrire une fonction **mini()** qui reçoit deux arguments et qui retourne le plus petit des deux. Pour éviter d'écrire autant de fonctions mini que de types à gérer, on va utiliser un ***template***. Ce sera le compilateur qui « générera le code pour chaque type utilisé » (ici **int** et **float**) :

```
#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
}
```

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**.

Par exemple dans l'exemple précédent, on ne peut pas utiliser la fonction *template* **`mini()`** avec le type **`char *`** (les chaînes de caractères en C n'admettent pas l'opérateur **`<`**). Dans ce cas là, on fera alors une spécialisation totale pour ce type :

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

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

Exemple n°2 : classe template

On désire réaliser une classe conteneur de type Tableau. Là encore, il faudrait réécrire cette classe pour chaque type à collecter. On va donc créer une classe ***template***. Ce sera le compilateur qui « générera le code pour chaque type utilisé » (ici **int** et **float**) :

```
// avec des méthodes inline
#include <iostream>
using namespace std;

template<class T> class Array
{
   private:
      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;   // un "tableau" d'entiers (ici T = int)
   Array<float> fa; // un "tableau" de réels (ici T = float)

   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;
}
```

Exemple n°3 : classe template avec des 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
{
   private:
      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;
   // ...
}
```

Avantages

Rappels : 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.

Inconvénients

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* → **`Array<int>`** par exemple).

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.

Solution possible

Il existe plusieurs solutions qui sont liées aux options des compilateurs en rapport aux *templates*.

Il est aussi possible de s'organiser de la manière suivante : la classe template sera déployée par **3 fichiers**

- **`templateArray.h`**

```
#ifndef TEMPLATEARRAY_H_
#define TEMPLATEARRAY_H_

#include <iostream>
using namespace std;

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

#include "templateArray.tcc"

#endif
```

- **templateArray.cc**

```
#include "templateArray.h"
// fichier vide : ne rien mettre d'autre ici
```

- **templateArray.tcc**

```
// Mettre ici les définitions des méthodes de la classe
template<class T> T& Array<T>::operator[](int index)
{
   if(index >= 0 && index < size) return A[index];
   else cerr << "Index out of range" << endl;
}
```

Remarque : 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*** *(Extrait du man g++)*

Exemple n°4 : un Array de Number

Évidemment, on peut utiliser ses propres types comme type d'un *template* :

```
#include <iostream>

using namespace std;

template<class T>
class Array
{
   private:
      enum { size = 10 };
      T A[size];
   public:
      T& operator[](int index)
      {
         if(index >= 0 && index < size)
            return A[index];
         else cerr << "Index out of range" << endl;
      }
      int length() const { return size; }
};

class Number
{
   private:
      float f;
```

```
   public:
      Number(float ff = 0.0f) : f(ff) { }
      Number& operator=(const Number& n)
      {
         f = n.f;
         return *this;
      }
      // surcharge de l'opérateur de cast
      operator float() const { return f; }
      friend ostream& operator<<(ostream& os, const Number& x)
      {
            return os << x.f;
      }
};

int main()
{
   Array<Number> fa; // un "tableau" de Number

   cout << fa[0] << endl;

   fa[0] = 1.5;
   cout << float(fa[0]) << endl;

   cout << fa.length() << endl;
}
```

Exemple n°5 : notion d'itérateur

Dans la bibliothèque standard C++, on trouve de nombreux ***templates*** (par exemple, les entrées/sorties, les chaînes de caractères ou les conteneurs vector, list, ...).

Voici l'ébauche d'une classe *template* **`Pile`** qui dispose d'un **itérateur**.

```
#include <iostream>

using namespace std;

template<class T>
class Pile
{
   private:
      enum { size = 10 };
      T stack[size];
      int top;
```

```
   public:
      Pile() : top(0) {}
      void push(const T& i) {
         if(top < size)
            stack[top++] = i;
         else cerr << "Too many push()es" << endl;
      }
      T pop() {
         if(top > 0)
               return stack[--top];
         else  cerr << "Too many pop()s" << endl;
      }
      int length() const { return size; }
      class PileIter;
      friend class PileIter;

      class PileIter
      {
         private:
            Pile& s;
            int index;

         public:
            PileIter(Pile& is) : s(is), index(0) {}
            T operator++() { // Prefix
               if(index < s.top)
                     return s.stack[++index];
               else  cerr << "iterator moved out of range" << endl;
            }
            T operator++(int) { // Postfix
               if(index < s.top)
                     return s.stack[index++];
               else  cerr << "iterator moved out of range" << endl;
            }
      };
};

int main()
{
   Pile<float> is;
   for(int i = 0; i < 5; i++)
      is.push(i*1.5);

   // Traverse avec un itérateur:
```

```
   Pile<float>::PileIter it(is);
   for(int j = 0; j < 5; j++)
      cout << it++ << endl;

   cout << is.length() << endl;
}
```