Les templates

Introduction

Les mécanismes vus jusqu'à présent nous permettent :

- de créer des nouveaux types de données
  - qui ont le même statut que les types prédéfinis en C++
  - qui sont complètement autonomes

- de récupérer du code, développé au préalable, lorsqu'on définit un nouveau type de données, soit par agrégation, soit par héritage

- de créer une hiérarchie de structure de données sous forme de container

Cependant, la hiérarchie qui a été mise en place ne concerne que les entiers : pile d'entiers, file d'entiers, …

Qu'en est-il si l'on veut gérer une pile de réels ou une file d'objets para ?

Dans l'état actuel des choses, il faut pratiquement tout réécrire ☹… Alors que l'un des buts de la programmation orientée objet est la **réutilisabilité**…

Prenons l'exemple de la file (rappel) :

```
class structure
{
  protected :
    int nbreElem;

  public :
    structure () { nbreElem = 0; }
    virtual int tete (void) = 0;
    virtual boolean empty (void);
    virtual boolean full (void);
    virtual boolean ajout (int n);
    virtual boolean retrait (void);
};

struct noeud {
  int valeur;
  noeud *suiv;
};

class liste : public structure
{
  protected :
    noeud *pListe;

  public :
    liste (void);
    ~liste (void);
    int tete (void);
    boolean empty (void);
    boolean full (void);
    void affContenu (void);
};

class file : public liste
{
  private :
    noeud *pFin;

  public :
    file (void);
    boolean ajout (int n);
    boolean retrait (void);
    int queue (void);
};
```

Donc, si on veut créer une file contenant des « float » à la place des « int », nous devons

- écrire une structure noeud adaptée aux float
- réécrire toutes les classes de la hiérarchie en remplaçant « int » par « float », avec des noms adaptés…

Ce qui donnerait :

File d'entiers	File de réels

`class structureInt`
`{`
`  protected :`
`  int nbreElem;`
 	
`  public :`
`  structure () { nbreElem = 0; }`
`  virtual int tete (void) = 0;`
`  virtual boolean empty (void);`
`  virtual boolean full (void);`
`  virtual boolean ajout (int n);`
`  virtual boolean retrait (void);`
`};`
 	
`struct noeudInt {`
`  int valeur;`
`  noeudInt *suiv;`
`};`
 	
`class listeInt : public structureInt`
`{`
`  protected :`
`  noeudInt *pListe;`
 	
`  public :`
`  liste (void);`
`  ~liste (void);`
`  int tete (void);`
`  boolean empty (void);`
`  boolean full (void);`
`  void affContenu (void);`
`};`
 	
`class fileInt : public listeInt`
`{`
`  private :`
`  noeudInt *pFin;`
 	
`  public :`
`  file (void);`
`  boolean ajout (int n);`
`  boolean retrait (void);`
`  int queue (void);`
`};`
	`class structureFloat`
	`{`
	`  protected :`
	`  int nbreElem;`
	
	`  public :`
	`  structure () { nbreElem = 0; }`
	`  virtual float tete (void) = 0;`
	`  virtual boolean empty (void);`
	`  virtual boolean full (void);`
	`  virtual boolean ajout (float n);`
	`  virtual boolean retrait (void);`
	`};`
	
	`struct noeudFloat {`
	`  int valeur;`
	`  noeudFloat *suiv;`
	`};`
	
	`class listeFloat : public`
	`structureFloat {`
	`  protected :`
	`  noeudFloat *pListe;`
	
	`  public :`
	`  liste (void);`
	`  ~liste (void);`
	`  float tete (void);`
	`  boolean empty (void);`
	`  boolean full (void);`
	`  void affContenu (void);`
	`};`
	
	`class fileFloat : public listeFloat`
	`{`
	`  private :`
	`  noeudFloat *pFin;`
	
	`  public :`
	`  file (void);`
	`  boolean ajout (float n);`
	`  boolean retrait (void);`
	`  float queue (void);`
	`};`

Ce qui est extrêmement lourd et inacceptable POO qui vise la réutilisabilité du code…

Le C++ permet l'**écriture de code** qui est **« générique »**, c'est-à-dire indépendant d'un type de données particulier

➔ cela est réalisable à l'aide des **« templates »**

Définition

La notion de **template** permet de créer en C++ des **classes** (ou des **fonctions**) **génériques**, c'est-à-dire **indépendantes d'un type de données**.

➔ Une **fonction « template »** est une fonction dont au moins un des paramètres est « générique », c'est-à-dire qu'il peut être d'un type quelconque

➔ Une **classe « template »** est une classe dont au moins une de ses variables membres est « générique », c'est-à-dire d'un type quelconque.

Instanciation des templates

Evidemment, puisque ces classes (fonctions) sont génériques, le compilateur ne crée pas de code lorsqu'on se contente de les définir. Il le fera lorsque l'on précisera quelle version de la classe (ou de la fonction) on désire utiliser dans notre programme.

C'est ce que l'on appelle l'**instanciation des templates**.

Les templates du C++

Définition d'une classe template

La syntaxe est la suivante :

```
template <typename nom_parametre [,class nom_parametre]>
class nom_classe
{
   …
};
```

Définition rapportée d'une méthode de template

La syntaxe est la suivante :

```
template <typename nom_parametre>
type_renvoye nom_classe<nom_parametre>::nom_methode(liste_arguments)
{
   …
}
```

Utilisation d'une classe template

Il faut préciser au compilateur le type de données qui va réellement être utilisé. On parle d'instanciation de template.

Instancier un template signifie donc que l'on demande au compilateur de générer le code de la classe en y remplaçant les types formels de la définition par les types effectivement utilisés.

L'instanciation d'une classe template se fait simplement en précisant le type de paramètre entre « pinces ».

Exemple : retour sur notre template « Vecteur »

Fichier **Templa1.cpp** :

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

template <typename T>
class Vecteur {          // Définition de la classe template Vecteur
  private :
    T *data;
    int nbElem;

  public :
    Vecteur (int n=10) {
      data = new T[n];
      nbElem = n;
    }
    ~Vecteur () {
      delete [] data;
    }
    T& operator[] (int i) {
      return *(data+i);
    }
};

template <typename T>   // Définition rapporté du constructeur
Vecteur<T>::Vecteur(int n) {
  data = new T[n];    // Appel du constructeur par defaut de T
  nbElem = n;
}

struct Coord {
  string nom;
  float salaire;
};

ostream& operator<<(ostream& s, const Coord& c) {
  s << c.nom << " --- " << c.salaire;
  return s;
}

int main()
{
  int ne;
  cout << "Combien d'elements ? ";
  cin >> ne;
```

```
  Vecteur<int> vi(ne); // instanciation du template pour int et
                       // instanciation de la variable vi
  for (int i=0; i<ne; i++) {
    vi[i] = i*(2*i+1); // utilisation operateur [] du template
  }
  cout << "Vecteur d'entiers ";
  for (int i=0; i<ne; i++) {
    cout << vi[i] << " / ";
  }
  cout << endl;

  Vecteur<char> vc(ne); // instanciation du template pour char et
                        // instanciation de la variable vc
  for (int i=0; i<ne; i++) {
    vc[i] = (char) i*(2*i+1) + 65;
  }
  cout << "Vecteur de caracteres ";
  for (int i=0; i<ne; i++) {
    cout << vc[i] << " / ";
  }
  cout << endl;

  Vecteur<Coord> vCoord(ne); // instanciation du template pour Coord
                             // et instanciation de la variable vCoord
  for (int i=0; i<ne; i++) {
    cout << "Nom : "; cin >> vCoord[i].nom;
    cout << "Salaire : "; cin >> vCoord[i].salaire;
  }
  for (int i=0; i<ne; i++) {
    cout << vCoord[i] << endl;
  }
  cout << endl;

  return 0;
}
```

dont un exemple d'exécution fournit

```
[student@moon Template]$ Templa1
Combien d'elements ? 4
Vecteur d'entiers 0 / 3 / 10 / 21 /
Vecteur de caracteres A / D / K / V /
Nom : wagner
Salaire : 2500
Nom : charlet
Salaire : 2800
Nom : caprasse
```

```
Salaire : 2100
Nom : madani
Salaire : 2200
wagner --- 2500
charlet --- 2800
caprasse --- 2100
madani --- 2200

[student@moon Template]$
```

Remarques

1. Une classe template peut **admettre plusieurs paramètres « génériques »**. Certains de ceux-ci peuvent même avoir une « valeur » par défaut.

Exemple :

```
template <typename CA,typename CB=coord,typename X=int>
class Machin { … };
```

2. **Une classe template peut être argument d'une autre classe template**.

Exemple : Soient deux classes template

```
template <typename X>
class List {…};

template <typename T>
class Set {…};
```

Alors, on pourrait imaginer d'instancier

```
List<Set<int>> listeDEnsembles;
```

3. On peut **spécialiser des templates** c'est-à-dire **définir une version particulière, pour un type de données particulier**.

Exemple :

Définition de la classe générique :

```
template <typename T>
class List{…} ;
```

Spécialisation de la version "caractères" :

```
class List<char> {…} ;
```

L'instanciation des templates

Instanciation automatique

Depuis la version 6.0 du C++, l'instanciation des templates nécessaires à la confection de l'exécutable final est automatique.

Le compilateur crée les instanciations nécessaires et les place, d'une manière ou d'une autre, dans une zone accessible aux différents modules constituant l'application.

Ce sera au linker de faire le tri.

Au final,

Instancier un template revient à remplacer le type « template » (T dans nos exemples ci-dessus ») par un type précis et connu, et compiler le code de la classe et/ou de la fonction template pour ce type précis.

Le souci est que l'on ne connait le type précis qu'au moment où on en a besoin. Il est donc impossible de compiler du code template avant de connaître le type de données pour lequel il va être utilisé.

Le C++ moderne utilise pour cela deux types de fichiers :

- un d'extension **.hpp** contenant la déclaration de la classe ou des méthodes templates
- un d'extension **.ipp** contenant les « implémentations » de la classe ou des méthodes déclarées dans le fichier **.hpp**

Aucun de ces 2 fichiers ne peut être compilés tel quel :

- Le fichier **.hpp** est inclus à l'endroit où on a besoin d'instancier un template tandis que
- le fichier **.ipp** est inclus par le fichier **.hpp** lui-même.

Exemple de notre template « Vecteur » en fichiers séparés :

Fichier Coord.h :

```
#ifndef COORD_H
#define COORD_H

#include <iostream>
using namespace std;

struct Coord {
  string nom;
  float salaire;
};

ostream& operator<< (ostream& s, const Coord& c);

#endif
```

Fichier Coord.cpp :

```
#include "Coord.h"

ostream& operator<< (ostream& s, const Coord& c) {
  s << c.nom << " --- " << c.salaire;
  return s;
}
```

Fichier Vecteur.hpp :

```
#ifndef VECTEUR_H
#define VECTEUR_H

template <typename T>
class Vecteur {
  private :
    T *data;
    int nbElem;

  public :
    Vecteur (int n=10);
    ~Vecteur ();
    T& operator[] (int i);
};

#include "Vecteur.ipp"

#endif
```

Fichier Vecteur.ipp :

```
template <typename T>
Vecteur<T>::Vecteur(int n) {
  data = new T[n];
  nbElem = n;
}

template <typename T>
Vecteur<T>::~Vecteur() {
  delete [] data;
}

template <typename T>
T& Vecteur<T>::operator[](int i) {
  return *(data+i);
}
```

Fichier Templa2.cpp

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

#include "Vecteur.hpp"
#include "Coord.h"

int main(void)
{
  int ne;
  cout << "Combien d'elements ? ";cin >> ne;

  Vecteur<int> vi(ne);
  for (int i=0; i<ne; i++) {
    vi[i] = i*(2*i+1);
  }
  cout << "Vecteur d'entiers ";
  for (int i=0; i<ne; i++) {
    cout << vi[i] << " / ";
  }
  cout << endl;

  Vecteur<char> vc(ne);
  for (int i=0; i<ne; i++) {
    vc[i] = (char) i*(2*i+1) + 65;
  }
  cout << "Vecteur de caracteres ";
  for (int i=0; i<ne; i++) {
    cout << vc[i] << " / ";
  }
  cout << endl;

  Vecteur<Coord> vCoord(ne);
  for (int i=0; i<ne; i++) {
    cout << "Nom : "; cin >> vCoord[i].nom;
    cout << "Salaire : "; cin >> vCoord[i].salaire;
  }
  for (int i=0; i<ne; i++) {
    cout << vCoord[i];
  }
  cout << endl;

  return 0;
}
```

La compilation peut alors s'effectuer selon

```
[student@moon Template]$ ls -l
total 24
-rw-r--r--. 1 student student  123 11 jui 15:34 Coord.cpp
-rw-r--r--. 1 student student  183 11 jui 15:36 Coord.h
-rw-r--r--. 1 student student 1310 11 jui 15:34 Templa1.cpp
-rw-r--r--. 1 student student  798 11 jui 15:36 Templa2.cpp
-rw-r--r--. 1 student student  233 11 jui 15:42 Vecteur.hpp
-rw-r--r--. 1 student student  237 11 jui 15:27 Vecteur.ipp
[student@moon Template]$ g++ Coord.cpp -c
[student@moon Template]$ g++ Templa2.cpp Coord.o -o Templa2
[student@moon Template]$
```

où on observe que **ni le fichier Vecteur.hpp, ni le fichier Vecteur.ipp n'a été compilé**.

L'exécution fournit le même résultat que précédemment.

Une liste template

On peut à présent envisager de modifier la classe liste, développée précédemment, afin de la rendre « générique ».

Attention que la **déclaration d'une classe template amie** d'une autre doit se faire d'une manière un peu particulière :

1. La classe amie, ici **Iterateur**, doit d'abord être déclarée :

```
template <typename type> class Iterateur;
```

2. Ensuite, la relation d'amitié doit être précisée comme d'habitude :

```
friend class Iterateur<T>;
```

Fichier Liste.hpp

```
#ifndef LISTE_H
#define LISTE_H

#include <iostream>
using namespace std;

template <typename T>
struct noeud {
  T valeur;
  noeud<T> *suiv;
};

// Declaration de la classe iterateur
template <typename T>
class Iterateur;

// Definition de la classe CListe
template <typename T>
class Liste {
  private :
    noeud<T> *pTete;

  public :
    Liste ();
    ~Liste ();
    bool isEmpty();
    T    head();
```

```
    void insert(const T& val);
    void removeHead();
    void removeTail();
    void display();

    friend class Iterateur<T>;
};

// Definition de la classe iterateur
template <typename T>
class Iterateur {
  private :
    Liste<T>& lis;
    noeud<T> *pCur;

  public :
    Iterateur(Liste<T>& l);
    bool end();
    void reset();
    bool operator++();
    bool operator++(int);
    operator T() const;
    T& operator&();
};

#include "Liste.ipp"

#endif
```

Fichier Liste.ipp

```
//***** Classe Liste ****************************************
template <typename T>
Liste<T>::Liste():pTete(nullptr) {
};

template <typename T>
Liste<T>::~Liste()
{
  noeud<T> *tmp;
  while (pTete != nullptr) {
    tmp = pTete->suiv;
    delete pTete;
    pTete = tmp;
  }
}
```

```
template <typename T>
bool Liste<T>::isEmpty() {
  return pTete == nullptr;
}

template <typename T>
T Liste<T>::head() {
  return pTete->valeur; // Appel du constructeur de copie de la
                        // classe T
}

template <typename T>
void Liste<T>::insert(const T& val) {
  noeud<T> *tmp = new noeud<T>();
  tmp->valeur = val;
  tmp->suiv = pTete;
  pTete = tmp;
}

template <typename T>
void Liste<T>::removeHead() {
  if (pTete != nullptr) {
    noeud<T> *tmp = pTete;
    pTete = pTete->suiv;
    delete tmp;
  }
}

template <typename T>
void Liste<T>::removeTail() {
  if (pTete == nullptr) return;
  if (pTete->suiv == nullptr) {
    delete pTete;
    pTete = nullptr;
    return;
  }
  noeud<T> *tmp = pTete;
  while (tmp->suiv->suiv != nullptr) {
    tmp = tmp->suiv;
  }
  delete tmp->suiv;
  tmp->suiv = nullptr;
}

template <typename T>
void Liste<T>::display() {
  Iterateur<T> it(*this);
  cout << "( ";
```

```
  while (!it.end()) {
    cout << (T)it << " , ";
    it++;
  }
  cout << "\x8\x8)" << endl;
}

//***** Classe Iterateur ****************************************
template <typename T>
Iterateur<T>::Iterateur(Liste<T>& l): lis(l),pCur(l.pTete) {
};

template <typename T>
bool Iterateur<T>::end() {
  return pCur == nullptr;
}

template <typename T>
void Iterateur<T>::reset() {
  pCur = lis.pTete;
}

template <typename T>
bool Iterateur<T>::operator++() {
  if (pCur) {
    pCur = pCur->suiv;
    return true;
  }
  return false;
}

template <typename T>
bool Iterateur<T>::operator++(int) {
  return operator++();
}

template <typename T>
Iterateur<T>::operator T() const {
  return pCur->valeur;
}

template <typename T>
T& Iterateur<T>::operator&() {
  return pCur->valeur;
}
```

Fichier Templa3.cpp

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

#include "Liste.hpp"
#include "Coord.h"

int main(void)
{
  Liste<int> l;
  for (int i=0; i<4 ; i++) l.insert (i*11);
  l.display();
  cout << "Destruction du dernier element: " << endl;
  l.removeTail();
  l.display();

  Iterateur<int> it(l);
  cout << "Examen de la liste" << endl;
  for (it.reset(); !it.end(); it++) {
    cout << "* " << (int) it << endl;
    cout << " --> nouvelle valeur ? "; int nb; cin >> nb;
    &it = nb;
  }
  l.display();

  Liste<Coord> lc;
  for (int i=0; i<3 ; i++) {
    Coord c;
    cout << "Nom : "; cin >> c.nom;
    cout << "Salaire : "; cin >> c.salaire;
    lc.insert(c);
  }
  lc.display();

  Iterateur<Coord> itc(lc);
  cout << "Examen de la liste" << endl;
  for (itc.reset(); !itc.end(); itc++) {
    cout << "* " << (Coord)itc << endl;
  }

  return 0;
}
```

dont la compilation et un exemple d'exécution fournit

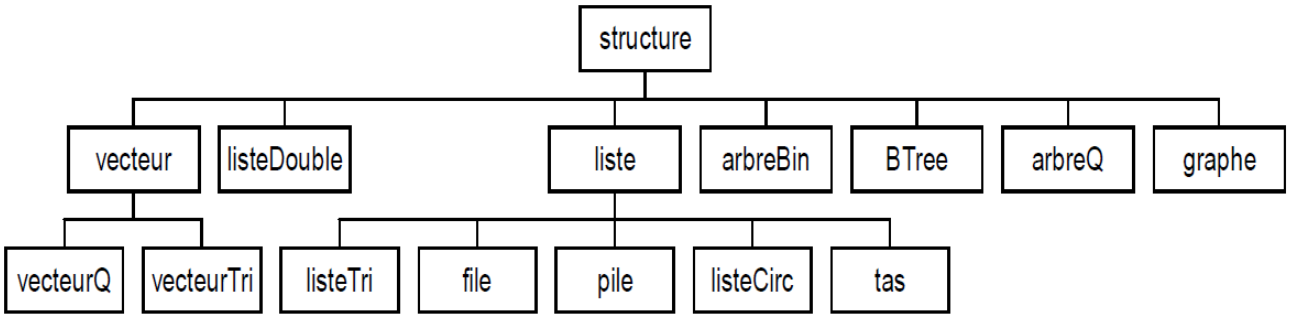
```
[student@moon Template]$ g++ Templa3.cpp Coord.o -o Templa3
[student@moon Template]$ Templa3
( 33 , 22 , 11 , 0 )
Destruction du dernier element:
( 33 , 22 , 11 )
Examen de la liste
* 33
 --> nouvelle valeur ? 3
* 22
 --> nouvelle valeur ? 2
* 11
 --> nouvelle valeur ? 1
( 3 , 2 , 1 )
Nom : wagner
Salaire : 2900
Nom : quettier
Salaire : 3100
Nom : leonard
Salaire : 2800
( leonard --- 2800 , quettier --- 3100 , wagner --- 2900 )
Examen de la liste
* leonard --- 2800
* quettier --- 3100
* wagner --- 2900
[student@moon Template]$
```

La hiérarchie générale de structures de données

On peut adapter l'ensemble de la hiérarchie de structures de données vue précédemment pour faire en sorte d'avoir une hiérarchie « générique » de structures de données.

Mais pour cela, il faut pouvoir **dériver une classe d'une classe template**, ce qui ne pose aucun problème.

La hiérarchie a toujours la même structure :



La classe Structure devient template :

```
template <typename type>
class structure {
  protected :
    int nbreElem;

  public :
    structure () { nbreElem = 0; }
    int getNbreElem (void) { return nbreElem; }
    virtual type head (void) = 0; // fonction virtuelle pure
    virtual boolean empty (void);
    virtual boolean full (void);
    virtual void insert (type);
    virtual void remove (type = 0);
};
```

Et ainsi de suite… Par exemple, la classe list, munie de son itérateur devient :

```
template <typename type> class CLisIter;

template <typename type>
class list : public structure<type> // Héritage de template
{
  protected :
    noeud<type> *pListe;

  public :
    list (void) : pListe(0) {};
    ~list (void);
    boolean empty(void) { return pListe ? false : true; }
    type head(void) { return pListe->value; }
    void insert(type n);
    void removeHead (void);
    void removeTail (void);
    void affContenu (void);
    friend class listIter<type>;
};

…
```

La suite est prévisible ☺…

Les fonctions templates

Une fonction peut aussi être générique, c'est-à-dire avoir au moins un paramètre dont le type est spécifié lors de l'utilisation de la fonction.

Syntaxe :

Une telle fonction doit être précédée de

```
template <typename nom_parametre [,class nom_parametre]>
```

Exemple :

```
template <typename T> void Swap(T& x,T& y);

int main() {
  int a=5, b=3;
  cout << "avant : a = " << a << " b = " << b << endl;
  Swap(a,b);
  cout << "apres : a = " << a << " b = " << b << endl;

  float m=5.4, n=9.6;
  cout << "avant : m = " << m << " n = " << n << endl;
  Swap(m,n);
  cout << "apres : m = " << m << " n = " << n << endl;

  return 0;
}

template <typename T>
void Swap(T& x,T& y) {
  T temp = x; // Appel du constructeur de copie de la classe T
  x = y;      // Appel de l'opérateur = de la classe T
  y = temp;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
avant : a = 5 b = 3
apres : a = 3 b = 5
avant : m = 5.4 n = 9.6
apres : m = 9.6 n = 5.4
bash-3.00$
```