Programmation Orientée Objet C++ — Episode #7

Généricité et templates

D'après *Le langage C++ — Bjarne Stroustrup*

Modèles (de fonctions et de classes)

- Support direct de la programmation générique :
  un type peut apparaître comme **paramètre** de la définition d'une classe ou d'une fonction
- Aucun lien imposé entre les différents types utilisés comme arguments

Un modèle connu

```
vector< <type> > v;
```

`type` peut être quelconque.

Modèles (de fonctions et de classes)

- Support direct de la programmation générique :
  un type peut apparaître comme **paramètre** de la définition d'une classe ou d'une fonction
- Aucun lien imposé entre les différents types utilisés comme arguments

`vector`, `list`, `map` sont des modèles

Modèles (de fonctions et de classes)

- Support direct de la programmation générique :
  un type peut apparaître comme **paramètre** de la définition d'une classe ou d'une fonction
- Aucun lien imposé entre les différents types utilisés comme arguments

La syntaxe
`template< <paramètres> >` **devant** déclaration et définition

Modèles (de fonctions et de classes)

- Support direct de la programmation générique :
  un type peut apparaître comme **paramètre** de la définition d'une classe ou d'une fonction
- Aucun lien imposé entre les différents types utilisés comme arguments

Présentation
Du plus simple (en apparence) au plus complexe...
... du moins au plus utile.

Un modèle de fonction simple

Les `template`

La syntaxe

`template < paramètres >` **devant** déclaration **et** définition

```
1  template < class T >
2  void echange(T &a, T &b) {
3    T tmp = a;
4    a = b;
5    b = tmp;
6  }
```

Un premier constat

Le type `T` doit posséder l'`operator=`.

En action

```cpp
#include <iostream>
#include <string>

template < class T >
void echange(T &a, T &b) {
  T tmp = a;
  a = b;
  b = tmp;
}

int main() {
  int a = 5; int b = 12;
  std::string s = "ceci"; std::string t = "echange";
  echange<int>(a,b);      // <int> optionnel
  echange(s,t);          // le type est déduit automatiquement
  echange(a,s);          // ?
  std::cout << a << "\t" << b << std::endl;
  std::cout << s << "\t" << t << std::endl;
}
```

Terminologie

Déclaration d'un modèle, et le type `T` apparaîtra en **argument** dans la déclaration.

Terminologie
`T` est un **paramètre de modèle**.

La **portée** de `T` s'étend jusqu'à la fin de la déclaration préfixée par `template`.

Remarque — faux-ami
`T` est un nom de **type**, et pas nécessairement d'une **classe**.

Un premier modèle de classe

Un modèle de `Pile`

```
1  #include <iostream>
2  #include <assert.h>
3  template< class C > class Pile {
4    private:
5      int taille_max;
6      int n;
7      C* pile;
8    public:
9      Pile(int _taille_max) : taille_max(_taille_max), n(0) {
10       (*this).pile = new C[taille_max];
11     }
12     C depiler() {
13       assert(n > 0);
14       n--; return pile[n];
15     }
16     void empiler(C _add) {
17       assert(n < taille_max);
18       pile[n] = _add; n++;
19     }
20 };
```

Pile de caractères

```cpp
#include "stack.hpp"

int main() {
  Pile<char> p(5);
  p.empiler('l');
  p.empiler('i');
  p.empiler('f');
  p.empiler('o');

  std::cout << p.depiler() << std::endl;

  p.empiler('f');
  p.empiler('i');
  p.empiler('f');

  return 0;
}
```

Pile de mots

```cpp
#include "stack.hpp"
#include <string>

int main() {
  Pile<std::string> p(5);
  p.empiler(std::string("ceci"));
  p.empiler(std::string("n'est"));
  p.empiler(std::string("pas"));
  p.empiler(std::string("une"));
  p.empiler(std::string("pile"));

  std::cout << p.depiler() << std::endl;

  return 0;
}
```

Quelques remarques

- Classes ordinaires
- *Aucun mécanisme d'exécution propre aux modèles*
- La quantité de code générée n'est pas forcément réduite
- *Presque* aussi optimal que du code généré à la main

Côté conception
Implémenter *correctement* une classe *particulière* avant de coder les mécanismes de `template`.

Séparer prototypes et implémentations ?

Attention
Ne pas séparer la déclaration d'un `template` (`.hpp`) de son implémentation (`.cpp`).

Toute l'information doit être disponible pour la compilation.

Séparer prototypes et implémentations ?

Une solution

Inclure un fichier `.tpp` en fin de header

```
1  #include <iostream>
2  #include <assert.h>
3
4  template< class C > class Pile {
5    private:
6      int taille_max;
7      int n;
8      C* pile;
9    public:
10     Pile(int _taille_max);
11     C depiler();
12     void empiler(C _add);
13 };
14
15 #include "stack_h.tpp"
```

Séparer prototypes et implémentations ?

Une solution

Inclure un fichier `.tpp` en fin de header

```
1  template< class C >
2  Pile<C>::Pile(int _taille_max) : taille_max(_taille_max), n(0) {
3        (*this).pile = new C[taille_max];
4  }
5
6  template< class C > C Pile<C>::depiler() {
7    assert(n > 0);
8    n--;
9    return pile[n];
10 }
11
12 template< class C > void Pile<C>::empiler(C _add) {
13   assert(n < taille_max);
14   pile[n] = _add;
15   n++;
16 }
```

Le constructeur

La syntaxe explicite du constructeur devrait être :

```
1 template< class C >
2 Pile<C>::Pile<C>(int _taille_max) : /* CODE */
```

Dans la portée de `Pile<C>`, la qualification avec `<C>` est redondante pour le nom du modèle lui-même.

Les paramètres de modèles

Paramètres acceptés

Paramètres de modèle
Ce sont les `class` : `template< class T >`.

Paramètres typés
Les paramètres précédents peuvent être utilisés :

```
template< class T, T def_val >
```

Paramètres de types ordinaires
```
template< class T, int i >
```

De l'utilité des paramètres ordinaires

```
1  template< class T, int i> class Buffer {
2    T v[i];
3    int taille;
4    public:
5      Buffer() : taille(i) {}
6      // ...
7  };
```

La déclaration

```
1  Buffer<char, 127> cbuf; // constante obligatoire
2  Buffer<Objet, 8> obuf; // constante obligatoire
```

De l'utilité des paramètres ordinaires

```
1  template< class T, int i> class Buffer {
2    T v[i];
3    int taille;
4    public:
5      Buffer() : taille(i) {}
6      // ...
7  };
```

Utilité

Efficacité d'exécution et concision (évite les `vector` ou `string`, parfois trop généraux).

De l'utilité des paramètres ordinaires

```
1  template< class T, int i> class Buffer {
2    T v[i];
3    int taille;
4    public:
5      Buffer() : taille(i) {}
6      // ...
7  };
```

Attention

Tous les types ordinaires ne peuvent pas être utilisés : `double`, `const char*`, ...

Un modèle plus complexe/connu

Modèle de chaîne

```
1  template<class C> class String {
2    private:
3      struct Srep;
4      Srep *rep;
5    public:
6      String();
7      String(const C*); // Utilisation du paramètre de template
8      String(const String&);
9
10     C read(int i) const; // Accesseur constant
11 };
```

Noms de classes depuis un modèle

```
1  String<char> cs;
2  String<unsigned_char> us;
3  String<wchar_t> ws;
4
5  class Jchar { // caractères japonais };
6  String<Jchar> js;
```

`String< n'importe quoi > ?`

Dans l'implémentation du modèle, certaines propriétés du type `C` peuvent être attendues.

Pour des caractères : comparaison, casse, ...

Noms de classes depuis un modèle

```
1  String<char> cs;
2  String<unsigned_char> us;
3  String<wchar_t> ws;
4
5  class Jchar { // caractères japonais };
6  String<Jchar> js;
```

Classe `string` de la STL ?

En fait : `typedef basic_string<char> string;`

Basée sur un **modèle** !

Noms de classes depuis un modèle

```
1  String<char> cs;
2  String<unsigned_char> us;
3  String<wchar_t> ws;
4
5  class Jchar { // caractères japonais };
6  String<Jchar> js;
```

Classe `string` de la STL ?

Le `typedef` raccourcit le nom mais aussi -surtout- cache l'utilisation (et donc l'existence) du modèle.

Définition (partielle) du modèle

```
template<class C> class String {
  private:
    struct Srep {
      C* s;
      int taille;
      ...
    };
    Srep *rep;
  public:
    String();
    String(const C*);
    String(const String&);

    C read(int i) const;
};
```

```
template<class C> C String<C>::
    read(int i) const { return
    rep->s[i]; }
```

Définition (partielle) du modèle

```
1  template<class C> class String {
2    private:
3      struct Srep {
4        C* s;
5        int taille;
6        ...
7      };
8      Srep *rep;
9    public:
10     String();
11     String(const C*);
12     String(const String&);
13
14     C read(int i) const;
15 };
```

```
1  template<class C> String<C>::
       String() {
2    rep = new Srep(0,C()); // ?
3  }
```

Un constructeur pour une structure ?

Différences `struct` et `class`

Plutôt *claires* en termes de concept...
moins évident côté implémentation.

Un début d'explication
Les `class` rendent leurs attributs et méthodes **privés** par défaut.

- Préférer `class` dès qu'il y a des méthodes.
- Utiliser `struct` comme un enregistrement du `C`.

Code généré

Uniquement ce qui est nécessaire :

```
1  #include "stack.hpp"
2
3  int main() {
4    Pile<char> p(5);;
5    p.empiler('l');
6    p.empiler('i');
7    p.empiler('f');
8    p.empiler('o');
9    return 0;
10 }
```

`depiler()` n'est pas générée.

Code généré

Uniquement ce qui est nécessaire :

```
1  #include <iostream>
2  template< class T >
3  T maximum(T const &a, T const &b) {
4    std::cout << "Template T" << std::endl;
5    return a < b ? b : a;
6  }
7  template< class T, class U>
8  T maximum(T const &a, U const &b) {
9    std::cout << "Template T, U" << std::endl;
10   return a < b ? b : a;
11 }
12 int main() {
13   maximum(1,2);
14 }
```

Seul le `Template T` est généré.

Schématiquement

À la compilation

- Déduction des types
- Vérification de la syntaxe, des types, de la sémantiques
- Recherche des bonnes fonctions et procédures

- Processus purement statique
- Création de code à chaque fois (lourd mais optimisé)
- Création du code nécessaire *uniquement*

Toute l'information nécessaire doit être disponible pour la compilation

- Impossible de mettre un patron dans un `.o`
- Aucun fichier `.cpp` — longs fichiers `.hpp`