Programmation Impérative C++ — Episode #8

Retour sur les `template`

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`

Les modèles de fonction

L'algorithme `sort`

```
template< class T > void sort(vector<T> &);
```

```
1  void f(vector<int> &vi, vector<string> &vs) {
2    sort(vi); // sort(vector<int> &);
3    sort(vs); // sort(vector<string> &);
4  }
```

Contraintes sur les types

```
1  #include <iostream>
2
3  template< class T, class U>
4  T maximum(T const &a, U const &b) {
5    std::cout << "Template T, U" << std::endl;
6    return a < b ? b : a;
7  }
8
9  int main() {
10   std::cout << maximum(1,2.6) << std::endl;
11 }
```

Contraintes sur les types

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

Contraintes sur les types

- A priori pas de contraintes sur `T` et `U`
- Dans les faits, besoin de :
  - `bool operator< (T,U)` pour le test
  - `U::operator()` pour la conversion

Arguments des modèles de fonction

Besoin
Déduire les arguments de modèles pour un appel à partir des arguments de fonction.

Pour le compilateur
Possibilité de déduire les types en argument à partir d'un appel, à condition que la liste d'arguments de la fonction identifie de façon **unique** l'ensemble des arguments du modèle.

Arguments des modèles de fonction

```
1  template< class T, int i > T& lookup(Buffer< T, i >& b, const
       char* p);
2
3  class Record {
4    const char v[12];
5    // ...
6  };
7
8  Record& f(Buffer<Record, 128>& buf, const char* p) {
9    return lookup(buf, p); // utilise lookup() où
10                 // T est un Record
11                 // et i est 128
12 }
```

Remarque
Les paramètres d'un modèle de classe ne sont jamais déduits.

Spécification explicite

Déduction impossible ?

Dans ce cas
Il faut le spécifier **explicitement**.

Une fonction sans arguments avec type de retour spécifié par `template`.

```
1  template< class T > class vector { /* ... */};
2  template< class T > T* create(); // construit un T et renvoie un
       pointeur de T
3
4  void f() {
5    vector< int > v; // classe, modèle 'int' en argument
6    int* p = create<int>(); // fonction, modèle 'int' en argument
7  }
```

Utilisation de la spécification explicite

Fournir un type renvoyé pour une fonction modèle

Utilisation de la spécification explicite

```
1  template< class T, class U > T implicit_cast(U u) { return u; }
2  void f(int i) {
3    // ERREUR : on ne peut pas déduire T
4    implicit_cast(i);
5    // T est un double ; U est un int
6    implicit_cast<double>(i);
7    // T est un char ; U est un double
8    implicit_cast<char, double>(i);
9    // T est un char* ; U est un int
10   implicit_cast<char*, int>(i);
11   // ERREUR : on ne peut pas convertir un int en char*
12 }
```

Remarque

Seuls les derniers arguments de la liste peuvent être omis de la liste des arguments de modèle explicite.

Utilisation de la spécification explicite

```
1  template< class T, class U > T implicit_cast(U u) { return u; }
2  void f(int i) {
3    // ERREUR : on ne peut pas déduire T
4    implicit_cast(i);
5    // T est un double ; U est un int
6    implicit_cast<double>(i);
7    // T est un char ; U est un double
8    implicit_cast<char, double>(i);
9    // T est un char* ; U est un int
10   implicit_cast<char*, int>(i);
11   // ERREUR : on ne peut pas convertir un int en char*
12 }
```

Exemple

Syntaxe de `static_cast` et `dynamic_cast` : fonction modèle explicitement spécifiée.

Contraintes sur les types

```
1  #include <iostream>
2
3  template< class T, class U>
4  T maximum(T const &a, U const &b) {
5    std::cout << "Template T, U" << std::endl;
6    return a < b ? b : a;
7  }
8
9  int main() {
10   std::cout << maximum(1,2.6) << std::endl;
11 }
```

Contraintes sur les types

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

`class` et limites d'un `template`

Restrictions
Nécessité d'être valide :

- à la déclaration (indépendamment des paramètres)
- à l'instantiation (pour ce qui est engendré) : les erreurs liées à un paramètre de modèle ne peuvent être détectée *avant* la première utilisation de ce modèle pour un argument particulier.

Erreur si `Rec` ne redéfinit pas `operator<<`.

`class` et limites d'un `template`

Mais encore ?

```
1  class List { /* ... */ }
2  class Rec { /* ... */ }
3
4  void f(List<int> &li, List<Rec> & lr) {
5    li.print();   // print utilise std::cout <<
6    lr.print();   // print utilise std::cout <<
7  }
```

Erreur si `Rec` ne redéfinit pas `operator<<`.

Surcharge de modèles de fonction

Surcharge de modèles de fonction

```
1  #include <iostream>
2  using namespace std;
3
4  template < class T >
5  T maximum(T const &a, T const& b) {
6    return a < b ? b : a;
7  }
8
9  int main() {
10   cout << maximum(3,5) << endl;
11   cout << maximum(6.2, 9.3) << endl;
12 }
```

Surcharge de modèles de fonction

```
1  #include <iostream>
2  using namespace std;
3
4  template < class T >
5  T maximum(T const &a, T const& b) {
6    return a < b ? b : a;
7  }
8
9  int main() {
10   cout << maximum(3,5) << endl;
11   cout << maximum(6.2, 9.3) << endl;
12   cout << maximum(2, 8.3) << endl;
13 }
```

Surcharge de modèles de fonction

```
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(2,8.3);
14 }
```

Quelle fonction choisir ?

- Mécanique proche de la résolution des surcharges de fonction.
- Recherche pour chaque modèle de la spécialisation correspondant le mieux à l'ensemble des arguments de fonction en :
  (i) enlevant celles pour lesquelles les arguments ne peuvent correspondre
  (ii) gardant celles nécessitant le moins de transtypage implicite
  (iii) gardant celles avec le moins de paramètres de modèle (= le plus d'arguments ordinaires)
  (iv) préférant les fonctions aux modèles de fonctions

Un exemple de résolution

```
template< class T > T sqrt(T);
template< class T > complex<T> sqrt(complex<T>);
double sqrt(double);
```

```
void f(complex<double> z) {
  sqrt(2);  // sqrt<int>(int)
  sqrt(2.0); // sqrt(double)
  sqrt(z);  // sqrt<double>(complex<double>)
}
```

Dans le dernier cas, préféré à `sqrt< complex<double> >(complex<double>)`.

Spécification des règles via arguments de modèle

Tri de chaînes

Concepts : la *chaîne*, le *type d'élément* et le *critère de tri*.

Où placer le critère ?

- Conteneur ?
- Type ?

L'algorithme doit s'exprimer en termes **généraux**.

Spécification des règles via arguments de modèle

Tri de chaînes

Concepts : la *chaîne*, le *type d'élément* et le *critère de tri*.

Où placer le critère ?

- Conteneur ?
- Type ?

```
1  template< class T, class C >
2  int compare( const String<T>& s, const String<T>& s2 ) {
3    for(int i = 0; i < s.length() && i < s1.length(); i++)
4      if(!C::eq(s[i],s1[i])) return C::lt(s[i], s1[i]) ? -1 : 1;
5    return s?length() != s1.length();
6  }
```

Spécification des règles via arguments de modèle

Tri de chaînes

Concepts : la *chaîne*, le *type d'élément* et le *critère de tri*.

Où placer le critère ?

- Conteneur ?
- Type ?

Les bonnes définitions de `C::eq()` et `C::lt()` permettent de refléter les propriétés de tri désirées.

Par l'exemple

```
template< class T> class
    Cmp {
  public:
    static int eq(T a, T
    b)
      { return a == b; }
    static int lt(T a, T
    b)
      { return a < b; }
};
```

```
void f(String<char> s, String<char> s1
    ) {
  compare< char, Cmp<char> >(s, s1);

}
```

Par l'exemple

```
class Literate {
  public:
    static int eq(char a,
     char b)
      { return a == b; }
    static int lt(char,
    char);
    // examen d'une table
     en fonction de la
    valeur du caractère
};
```

```
void f(String<char> s, String<char> s1
    ) {
  compare< char, Cmp<char> >(s, s1);
  compare< char, Literate >(s, s1);
}
```

Les paramètres de modèle par défaut

Contrainte : spécifier un critère à chaque appel.

Via surcharge

```
1  // Comparaison à l'aide de C
2  template< class T, class C >
3  int compare( const String<T> &s, const String<T>& s1);
4
5  // Comparaison à l'aide de Cmp<T>
6  template< class T >
7  int compare( const String<T> &s, const String<T>& s1);
```

Les paramètres de modèle par défaut

Contrainte : spécifier un critère à chaque appel.

Via paramètres par défaut

```
1  template< class T, class C = Cmp<T> >
2  int compare( const String<T> s, const String<T> s1) {
3    /* ... */
4  }
```

Spécialisation de `template`

Spécialisation de fonctions modèle

Spécialisation de fonctions modèle

Algorithme de tri

Implémentation simple : utilise l'opérateur <.

```
template< class T > bool less(T a, T b) { return a < b; }
```

```
template< class T > void sort(Vector<T> &v) {
  /* ... */
  if(less(v[i], v[j]) )
    /* ... */
}
```

Mauvais tri de `vector<char*>`.

Spécialisation de fonctions modèle

Algorithme de tri

Implémentation simple : utilise l'opérateur <.

```
template<> bool less<const char*>(const char* a, const char* b)
  { return strcmp(a,b) < 0; }
```

```
template< class T > void sort(Vector<T> &v) {
  /* ... */
  if(less(v[i], v[j]) )
    /* ... */
}
```

Mieux : spécialisation du `template` pour `const char*`.

Spécialisation de fonctions modèle

Algorithme de tri

Implémentation simple : utilise l'opérateur <.

```
template<> bool less<const char*>(const char* a, const char* b)
  { return strcmp(a,b) < 0; }
```

```
template< class T > void sort(Vector<T> &v) {
  /* ... */
  if(less(v[i], v[j]) )
    /* ... */
}
```

`template<>` : la spécialisation peut être indiquée sans paramètre de modèle.

Spécialisation de fonctions modèle

Algorithme de tri

Implémentation simple : utilise l'opérateur <.

```
template<> bool less<const char*>(const char* a, const char* b)
  { return strcmp(a,b) < 0; }
```

```
template< class T > void sort(Vector<T> &v) {
  /* ... */
  if(less(v[i], v[j]) )
    /* ... */
}
```

`<const char*>` : la spécialisation doit être utilisée lorsque l'argument modèle est `const char*`.

Spécialisation de fonctions modèle

Algorithme de tri

Implémentation simple : utilise l'opérateur <.

```
template<> bool less(const char* a, const char* b)
  { return strcmp(a,b) < 0; }
```

```
template< class T > void sort(Vector<T> &v) {
  /* ... */
  if(less(v[i], v[j]) )
    /* ... */
}
```

Peut être déduit : inutile !

Spécialisation de classe

Spécialisation de classe

Par défaut : une seule définition.
Utilisée pour chaque (combinaison d')argument de modèle imaginable.

Problèmes

- Comment spécifier des implémentations alternatives pour des utilisations différentes d'une interface commune ?
- Comment restreindre l'utilisation du modèle (erreur *sauf si...*) ?

Solutions

Définitions de modèle différentes — choix de la plus appropriée laissé au compilateur.

Spécialisation et modèle général — `List`

- Une spécialisation pour toute liste de pointeurs (modèle différent sinon)
- Modèle général **déclaré** avant toute spécialisation.

```
1  template< class T > class List<T*> { /* ... */ }
2  template< class T > class List { /* ... */ }
3  /* ERREUR : modèle général après la spécialisation */
```

Spécialisation et modèle général — `List`

- Une spécialisation pour toute liste de pointeurs (modèle différent sinon)
- Modèle général **déclaré** avant toute spécialisation.

Information du modèle général
Paramètres de modèle à fournir pour l'utiliser (ou une spécialisation).

Spécialisation et modèle général — `List`

- Une spécialisation pour toute liste de pointeurs (modèle différent sinon)
- Modèle général **déclaré** avant toute spécialisation.

```
1  template< class T > class List;
2  template< class T > class List<T*> { /* ... */ };
3  /* La DECLARATION est suffisante */
```

Spécialisation et modèle général — `List`

- Une spécialisation pour toute liste de pointeurs (modèle différent sinon)
- Modèle général **déclaré** avant toute spécialisation.

```
1  template< class T > class List { /* ... */ };
2  List<int*> li;
3  template< class T > class List<T*> { /* ... */ };
4  /* ERREUR */
```

Spécialisation de classes

Utilisations probables d'un `Vector`

```
template< class T> class Vector
    {
  private:
    T* v;
    int taille;
  public:
    Vector();
    Vector(int);
    T& at(int i) { return v[i];
    }
    T& operator[](int i);
    /* ... */
};
```

```
Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

Beaucoup de pointeurs.

Type le plus commun ?

Les `vector` seront **souvent** d'un type pointeur.

Préserver un comportement polymorphique à l'exécution

Par défaut
Duplication de code pour les modèles.

Attention
Peut devenir un problème.

Une solution

Les conteneurs de pointeur peuvent partager une **unique** implémentation.

Spécialisation — `void*`

```
1  template<> class Vector<void *> {
2    void **p;
3    // ...
4    void*& operator[](int i);
5  };
```

Implémentation commune pour tous les `Vector` de pointeurs.

Une solution

Les conteneurs de pointeur peuvent partager une **unique** implémentation.

Spécialisation — `void*`

```
1  template<> class Vector<void *> {
2    void **p;
3    // ...
4    void*& operator[](int i);
5  };
```

`template<>` : spécialisation pouvant être indiquée sans paramètre de modèle.

Une solution

Les conteneurs de pointeur peuvent partager une **unique** implémentation.

Spécialisation — `void*`

```
1  template<> class Vector<void *> {
2    void **p;
3    // ...
4    void*& operator[](int i);
5  };
```

Les arguments de modèle pour lesquels la spécialisation doit être utilisée sont indiqués entre les <> situés après le nom : ici : `void*`

Quand et comment utiliser la spécialisation ?

Spécialisation complète
Aucun paramètre de modèle ne permet de spécifier (ou de déduire) à quel moment utiliser la spécialisation.

Seule utilisation
```
Vector< void* > vpv;
```

Quand et comment utiliser la spécialisation ?

Spécialisation complète

Aucun paramètre de modèle ne permet de spécifier (ou de déduire) à quel moment utiliser la spécialisation.

```
1  template< class T > class Vector<T*> : private Vector<void*> {
2    public:
3      typedef Vector<void*> Base;
4
5      Vector() : Base() {}
6      explicit Vector(int i) : Base(i) {}
7
8      // ...
9  }
```

Quand et comment utiliser la spécialisation ?

Spécialisation complète
Aucun paramètre de modèle ne permet de spécifier (ou de déduire) à quel moment utiliser la spécialisation.

`explicit ?` — interdit la conversion implicite

Spécialisation partielle

```
1  template< class T > class Vector<T*> : private Vector<void*> {
2    public:
3      typedef Vector<void*> Base;
4
5      Vector() : Base() {}
6      explicit Vector(int i) : Base(i) {}
7
8      // ...
9  }
```

Le paramètre de modèle est déduit *à partir* du modèle de spécialisation :

Dans `Vector<Shape*>`, `T` est `Shape` et non `Shape*`.

Spécialisation partielle

```
1  template< class T > class Vector<T*> : private Vector<void*> {
2    public:
3      typedef Vector<void*> Base;
4
5      Vector() : Base() {}
6      explicit Vector(int i) : Base(i) {}
7
8      // ...
9  }
```

Implémentation partagée pour tous les `Vector` de pointeurs.

Spécialisation partielle

```
1  template< class T > class Vector<T*> : private Vector<void*> {
2    public:
3      typedef Vector<void*> Base;
4
5      Vector() : Base() {}
6      explicit Vector(int i) : Base(i) {}
7
8      // ...
9  }
```

Perfectionnement de `Vector` obtenu sans affecter l'interface utilisateur.

L’ordre des spécialisations

Notion d’ordre — spécialisations `S` et `S'`
`S` est *plus spécialisée* que `S'` lorsque chaque liste d’arguments correspondant à `S` correspond à `S'`, et l’inverse est faux.

Par l’exemple

```
1 template< class T > class Vector;       // général
2 template< class T > class Vector<T*>;    // spécialisé pour tout
      pointeur
3 template<> class Vector<void*>;        // spécialisé pour void*
```

Dans la STL

Spécialisation de `vector< bool >` :

```
template<
  class T,
  class Allocator = std::allocator<T>
> class vector;
```

```
template<class Allocator>
class vector<bool, Allocator>;
```