École Nationale Supérieure d'Ingénieurs de Caen

6, Boulevard Maréchal Juin
F-14050 Caen Cedex, FRANCE

Template programming C++

Niveau	$3^{\text{ème}}$ **année**
Parcours	**Électronique et Physique Appliquée**
Unité d'enseignement	2E1AC2 **- Architectures pour le calcul** [**Parallélisme**]
Création	9 **Novembre** 2020
Responsables	**Emmanuel Cagniot** **emmanuel.cagniot@ensicaen.fr** **Hugo Descoubes** **Hugo.Descoubes@ensicaen.fr**

Table des matières

1 Généricité

La généricité (ou polymorphisme de compilation) désigne la possibilité de paramétrer une fonction, une méthode ou toutes les méthodes d'une classe par des types abstraits, des constantes entières, des alias ou encore des pointeurs, l'instanciation étant réalisée dès la compilation.

Ce type de polymorphisme permet de réduire la taille des codes puisqu'une même fonction, méthode ou classe est définie une fois pour toutes sous forme générique puis instanciée autant de fois que nécessaire. De plus, il est particulièrement utile dans les applications pour lesquelles le temps de calcul est un facteur crucial puisqu'il permet de se substituer au polymorphisme d'exécution.

La généricité ne s'arrête pas à la définition de fonctions méthodes ou classes génériques mais également à ce que l'on appelle la méta-programmation c'est à dire la possibilité d'écrire un programme capable de se ré-écrire lui-même.

La bibliothèque standard illustre les possibilités de ce type de programmation. Bien que certainement l'aspect le plus puissant du C++, le polymorphisme de compilation est également le moins connu et le plus sous-utilisé. Il est vrai que la syntaxe d'un méta-programme C++ est particulièrement abominable.

1.1 Fonctions génériques

La figure 1 présente un exemple de fonction générique. Dans ce dernier, nous définissons une fonction `appartient` paramétrée par le type abstrait `T`. Celle-ci prend comme arguments un tableau de type `T`, un alias constant pour la taille de ce tableau et un autre pour un élément de type `T`, puis indique si l'élément appartient au tableau.

```
1 template< typename T >
2 bool
3 appartient(const T tableau[], const unsigned long& taille, const T& elt) {
4   unsigned long i;
5   for (i = 0; i < taille && elt != tableau[i]; i ++);
6   return i != taille;
7 }
```

FIGURE 1 – Fonction générique `appartient`.

Comme une fonction classique, une fonction générique peut être déclarée puis définie ou définie directement dans le code source du programme principal. Cependant, dans le cas d'un module de bibliothèque, nous dirons pour simplifier que la fonction doit obligatoirement être définie en ligne, c'est à dire directement dans son fichier en-tête. Signalons toutefois que la norme propose un mécanisme permettant, comme pour une fonction classique, de la déclarer dans son fichier en-tête et de la définir dans son fichier source. Cependant, les problèmes posés par ce découplage sont tels que de nombreux compilateurs ne le propose pas.

La déclaration et la définition d'une fonction générique sont précédées d'une séquence `template<` $\texttt{composant}_1$`,` $\texttt{composant}_2$`, ...,` $\texttt{composant}_n$ `>` appelée patron (ou template). Les composants de cette séquence, séparés par des virgules, peuvent être des types abstraits, des constantes entières, des alias ou des pointeurs mais pas des nombres pseudo réels ou des instances de classes. Ces composants sont instanciés à la compilation et forment l'environnement de la fonction. Un type abstrait est représenté par le mot-clé `typename` suivi d'un nom. Ainsi, le patron `template< typename T, T valeur >` signifie que la fonction qui suit est paramétrée par le type abstrait `T` et une constante `valeur` de ce type.

La définition d'une fonction générique induit des contraintes sur les composants de son patron. Ainsi, dans l'exemple de la figure 1, l'expression `elt != tableau[i]` impose que le type abstrait `T` dispose de l'opérateur d'inégalité (`!=`). Dans le cas d'une instanciation par un type fondamental, cette condition ne pose aucun problème. Dans le cas d'une instanciation par une classe, cette dernière doit surcharger l'opérateur concerné.

La figure 2 présente différentes instanciations dites « implicites » de la fonction `appartient`, c'est à dire que le compilateur infère le type ou la valeur des composants du patron à partir de celui des arguments fournis à la fonction. Ainsi, dans le cas de l'appel `appartient(entier, taille, 4)`, le compilateur infère que le type abstrait `T` doit être instancié par le type fondamental `int`. La situation est plus compliquée dans le cas de l'appel `appartient(chaines, taille, std::sting("deux"))` puisque nous sommes obligés de convertir explicitement la constante littérale chaîne de caractères `"deux"` en une instance de classe `string`, le type par défaut associé à ces constantes étant `const char*`. Omettre cette conversion se solde par une erreur de compilation puisqu'aucun patron ne convient pour les types inférés par le compilateur. Une fonction générique peut également être instanciée explicitement. Ainsi, dans le cas de la fonction `appartient`, nous pouvons écrire `appartient< std::string >(chaines, taille, "deux")`, ce qui force le compilateur à utiliser la seconde version de la fonction en effectuant les conversions implicites adéquates entre le type des arguments et celui des paramètres formels.

```
1  const unsigned long taille         = 4
2
3  const int entiers[taille]          = { 1, 2, 3, 4 };
4  const char caracteres[taille]      = { 'a', 'b', 'c', 'd' };
5  const std::string chaines[taille] = { "une", "deux", "trois", "quatre" };
6
7  bool present;
8  present = appartient(entiers,     taille, 3);
9  present = appartient(caracteres, taille, 'h');
10 present = appartient(chaines,     taille, std::string("deux"));
```

FIGURE 2 – Différentes instanciations (implicites) de la fonction générique `appartient` de la figure 1.

L'instanciation d'une fonction se traduit, dans le code source, par la définition d'une nouvelle surcharge de la fonction avec les types ayant servi à l'instancier.

1.1.1 Spécialisation

La fonction générique `appartient` de la figure 1 est sémantiquement incorrect en cas d'instanciation par un type pseudo réel, l'opérateur d'inégalité n'ayant alors aucune légitimité (deux nombres pseudo réels ne peuvent être égaux qu'à un epsilon près). Pour corriger ce problème, il est nécessaire de spécialiser la fonction `appartient` pour ces types particuliers.

Dans le cas d'une fonction générique, la norme n'autorise qu'une spécialisation totale (et non partielle). Cette spécialisation prend la forme d'une nouvelle surcharge dans laquelle tous les composants abstraits du patron sont remplacés par des composants concrets (types fondamentaux ou classes, constantes entières, alias ou pointeurs). Le patron de cette fonction est alors vide (`template<>`) tandis que les composants concrets utilisés pour la spécialisation sont énumérés dans une séquence `< composant`$_1$`, composant`$_2$`, ..., composant`$_n$ `>` placée derrière le nom de la fonction et apparaissent tels quels dans son prototype et sa définition. La figure 3 présente la spécialisation de la fonction `appartient` de la figure 1 pour le type fondamental `float`.

Lorsque le mécanisme des spécialisations est mis en œuvre, l'ordre dans lequel sont définies les différentes spécialisations n'est pas anodin. Par conséquent, lorsque le nombre de ces spécialisations est

```
template<>
bool appartient< float >(const float tableau[],
                         const unsigned long& taille,
                         const float& elt) {
  constexpr float epsilon = 0.00001;
  unsigned long i = 0;
  while (i < taille) {
    const float& autre = tableau[i];
    const float ratio = std::fabs((elt - autre) / elt);
    if (ratio > epsilon) {
      i ++;
      continue;
    }
    return true;
  }
  return false;
}
```

FIGURE 3 – Spécialisation de la fonction générique `appartient` de la figure 1 pour le type fondamental `float`.

important, il est préférable d'utiliser l'instanciation explicite pour invoquer la bonne version (afin d'éviter que le compilateur ne mette en œuvre des conversions implicites de types pour déterminer quelle version invoquer). Cette règle est en fait celle qui prévaut lorsque le nombre de surcharges d'une fonction classique (non générique) est important (toujours invoquer avec les bons types pour ne pas laisser le compilateur choisir à notre place). Notons enfin qu'une fonction classique peut elle-même surcharger une fonction générique totalement spécialisée. Dans ce cas, la seconde sera utilisée uniquement si la première ne convient pas.

1.2 Méthodes génériques d'une classe non générique

Une classe non générique peut comporter des méthodes de classe (mot-clé `static`) ou d'instances génériques. Comme pour les fonctions, nous dirons que de telles méthodes doivent être définies en ligne.

Une méthode générique d'instance d'une classe non générique n'est pas redéfinissable dans les classes dérivées. Cependant, il est relativement facile de contourner cette limitation en :

1. surchargeant la méthode générique dans la classe dérivée ;
2. définissant une méthode non générique redéfinissable dans la classe de base, celle-ci invoquant la méthode générique de sa classe ;
3. redéfinissant la méthode non générique dans la classe dérivée afin qu'elle invoque la méthode générique surchargée de cette classe.

Les méthodes d'instance génériques d'une classe non générique n'étant pas redéfinissables, elle ne peuvent donc être abstraites.

Les méthodes génériques d'une classe non générique peuvent être instanciées implicitement ou explicitement.

1.2.1 Spécialisation

Comme les fonctions, les méthodes génériques d'une classe non génériques peuvent être spécialisées. Cependant, la norme impose des contraintes à ces spécialisations :

1. une spécialisation partielle peut être définie dans la déclaration de la classe ou à l'extérieur ;

2. une spécialisation totale ne peut être définie qu'à l'extérieur ;
3. les spécialisations appartiennent au même espace de noms que leur classe.

1.3 Classes génériques

En règle générale, une classe est dite générique si elle comporte au moins un attribut de classe (mot-clé `static`) ou d'instance d'un type abstrait `T`. Cependant, lorsqu'une classe non générique dépourvue d'attribut propose un ensemble de méthodes génériques ayant un commun le type `T`, cette classe peut être convertie en une classe générique paramétrée par le type `T`. Dans la pratique, ces méthodes génériques sont généralement des méthodes de classe représentant des algorithmes (exemple des classes `Arrays` ou `Collections` du langage JAVA).

Toutes les méthodes proposées par une classe générique sont par essence des méthodes génériques. Cependant, il faut introduire une distinction entre ces méthodes :

1. une méthode sera dite générique si son prototype ne mentionne pas de patron. Dans ce cas, son patron est implicitement celui de sa classe ;
2. une méthode sera dite sur-générique si son prototype mentionne un patron. Dans ce cas, son patron est constitué implicitement de celui de sa classe augmenté de celui mentionné dans son prototype.

Contrairement aux fonctions et aux méthodes génériques d'une classe non-générique, les classes génériques ne peuvent être instanciées qu'explicitement.

1.3.1 Spécialisation

Comme les méthodes génériques d'une classe non générique, les classes génériques peuvent être spécialisées partiellement ou totalement.

La norme n'autorise la définition d'un type synonyme (constructeur `typedef`) que pour des classes totalement spécialisées. Ainsi, si `Classe` représente une classe générique paramétrée par les types abstraits `T1` et `T2`, nous pouvons écrire :

```
typedef Classe< bool, int > MaClasse;
```

La norme propose néanmoins un autre mécanisme permettant de définir des synonymes pour des classes partiellement spécialisées via le mot clé `using`. Ainsi, nous pouvons écrire :

```
template< typename T1 >
using MaClasse = Classe< T1, int >;
```

1.3.2 Héritage

Une classe générique peut servir de classe de base. Cependant, si `Base` représente une classe générique paramétrée par un type abstrait et que `DeriveeT1` et `DeriveeT2` dérivent de `Base` tout en l'instanciant avec les types fondamentaux ou classe `T1` et `T2`, les classes `DeriveeT1` et `DeriveeT2` n'ont aucun ancêtre commun et donc aucun lien de parenté. Pour assurer un ancêtre commun à ces deux classes, il faudrait que la classe générique `Base` dérive elle-même d'une classe non générique.

Les méthodes génériques d'instance sont redéfinissables dans les classes dérivées tandis que les méthodes d'instance sur-génériques ne le sont pas. La technique permettant de contourner cette limitation est celle évoquée précédemment et concernant les méthodes génériques proposées par des classes non génériques. Les méthodes génériques d'instance étant redéfinissables, elles peuvent être abstraites et, par conséquent, leur classe également.

1.4 Paradigme de la méta-programmation

On appelle méta-programme un code capable de se ré-écrire ou de ré-écrire d'autres codes au moment de la compilation. La méta-programmation (ou *template programming* en C++ ou D) est certainement l'aspect le plus interéssant du polymorphisme de compilation puisque ce paradigme permet de résoudre un problème au moment de la compilation et non de l'exécution. Par conséquent, il s'agit d'un outil d'optimisation particulièrement puissant qui permet d'effectuer des calculs critiques dès la compilation. Cependant, il est vrai que la syntaxe à priori abominable (au premier abord uniquement) des méta-programmes rend ce type de programmation particulièrement rebutant pour les néophytes.

La spécialisation est le premier réflexe de tout méta-programmeur. Cependant, lorsque le nombre de composants d'un patron dépasse trois, le nombre de spécialisations potentielles (partielles et totales) explose. Il existe donc des idiomes (design patterns liés à un langage particulier) dédiés à la résolution de ce type de problèmes, les plus connus étant les idiomes de la `Trait Class`, celui du `Policy Class Design`, etc.

1.4.1 Fonction factorielle

L'implémentation récursive de la fonction factorielle dans le paradigme de la méta-programmation est un exemple simple représentant une bonne entrée en matière. La figure 4 présente la définition du template récursif `Fact`.

```
1  template < unsigned n >
2  class Fact {
3  public:
4    enum : unsigned long long { value = n * Fact< n - 1 >::value };
5  }; // Fact< n >
6
7  template<>
8  class Fact< 0 > {
9  public:
10   enum : unsigned long long { value = 1 };
11 }; // Fact< 0 >
```

FIGURE 4 – Template récursif représentant la fonction factorielle.

Dans cet exemple, la forme générale du template `Fact< n >` calcule récursivement (à la compilation) la valeur $n!$ comme $n! = n \times (n-1)!$ tandis que la spécialisation totale `Fact< 0 >` représente la condition d'arrêt de la récursion ($0! = 1$). Les différentes valeurs sont représentées par des constantes symboliques d'un type énuméré faiblement typé anonyme. Notons que pour des raisons un peu trop longues à détailler ici (les constantes symboliques des types énumérés ne sont pas des lvalues, contrairement aux constantes de classes), il faut préférer les `enum` aux constantes de classes (mot-clé `static`).

L'application de la fonction factorielle à une constante littérale entière (au moment de la compilation) se traduit par l'instanciation du template `Fact`, c'est à dire `auto f5 = Fact< 5 >::value`.

1.4.2 Somme des entiers d'une liste non vide de taille variable

Nous enchaînons avec un exemple un peu plus compliqué mettant en œuvre un apport majeur de la norme 2011 : les patrons de taille variable (variadic templates). Un patron de taille variable peut contenir un nombre arbitraire de composants de toute nature (type abstrait, constante entières, pointeurs ou alias), éventuellement aucun. Par conséquent, l'accès aux différents composants de ce patron se fait de manière récursive, tout comme l'accès aux éléments d'une liste dans le paradigme de la programmation fonctionnelle. La figure 5 présente le template récursif `SommeListe` permettant de calculer la somme des

entiers d'une liste non vide de taille variable.

```
1  template< int ... liste >
2  class SommeListe;
3
4  template< int car, int ... cdr >
5  class SommeListe< car, cdr ... > {
6  public:
7    enum : long { value = car + SommeListe< cdr ... >::value } ;
8  }; // SommeListe< car, cdr >
9
10 template< int car >
11 class SommeListe< car > {
12 public:
13   enum : long { value = car };
14 }; // SommeListe< car>
```

FIGURE 5 – Template récursif `SommeListe` permettant de calculer la somme des entiers d'une liste non vide de taille variable.

La syntaxe d'un composant de taille variable peut être `typename ... composant` ou bien encore `int ... composant`. Dans le premier cas, `composant` désigne une liste de taille variable pouvant comporter des entitées hétérogènes (types abstraits, constantes entières, pointeurs ou alias). Dans le second cas, `composant` désigne une liste de taille variable comportant des entités homogènes, ici des constantes entières. L'opérateur `...` placé devant le nom d'un composant sert à déclarer ce composant et à indiquer au compilateur que ce dernier est sous forme « compressée » (taille inconnue).

Dans l'exemple du template `SommeListe` de la figure 5, nous commençons par déclarer une classe `SommeListe` paramétrée par un composant de taille variable représentant une liste d'entiers. Nous spécialisons ensuite cette classe.

La spécialisation la plus simple concerne la liste ne contenant qu'un seul et unique entier (`SommeListe< car >`).

La spécialisation la plus intéressante concerne une liste comportant au moins deux entiers (`SommeListe< car, cdr ... >`). Nous adoptons ici les conventions des langages fonctionnels tels que LISP ou SCHEME dans lesquels la fonction `car` appliquée à une liste retourne son premier élément tandis que la fonction `cdr` appliquée à cette même liste en retourne le reste. Dans cette spécialisation, nous utilisons la syntaxe `cdr ...` qui désigne cette fois la forme « décompressée » du composant (l'opérateur `...` est ici placé derrière le nom du composant), c'est à dire une forme dans laquelle apparaît explicitement, en quelque sorte, chacun de ses éléments. Comme dans le cas de la fonction factorielle de l'exemple précédent, la constante symbolique `value` est initialisé à partir d'une expression qui instancie récursivement la classe `SommeListe` avec le `cdr` de sa liste et ce jusqu'à atteindre la condition d'arrêt correspondant à la liste ne contenant qu'un seul élément.