1 Les templates

Les templates sont un mécanisme puissant de factorisation de code, qui permettent d'écrire du code générique s'appliquant à des données, indépendamment de leur type. Plus précisément, ils permettent de produire en un seul fichier une famille de fonctions ou une famille de classes indicées par un type abstrait ou par un autre paramètre comme un entier.

1.1 Templating par un type, l'exemple des fonctions

Les templates sont issus originellement d'un souci de simplification de code par la factorisation, afin d'éviter les redondances. Prenons l'exemple d'une fonction Max qui prenne en argument un tableau de double et la taille de ce tableau :

```
1 double Max(double array[], int length)
2 {
3     double vmax = array[0];
4     for (int i = 1; i < length; i++)
5         if (array[i] > vmax)
6             vmax = array[i];
7     return vmax;
8 }
```

Si nous voulions définir la même fonction sur un tableau d'entiers, nous devrions alors produire le code suivant :

```
1 int Max(int array[], int length)
2 {
3     int vmax = array[0];
```

```
4       for (int i = 1; i < length; i++)
5           if (array[i] > vmax)
6               vmax = array[i];
7       return vmax;
8   }
```

De la même manière, nous pourrions définir la fonction Max sur un tableau de float, de ushort, de uint, de long, etc... Dans le cadre d'exemples plus complexes, le dédoublement du code pour chaque type est très pénalisant : tout d'abord, il nuit à la clarté du code pour le lecteur, mais il entraine aussi un risque important de divergence des différentes versions du code. En effet, si la sémantique de la fonction Max est incorrecte, elle le sera à la fois pour la version du code manipulant des entiers comme pour celle manipulant des double. Si un développeur est amené à améliorer ou débugger une version de cette fonction, il court le risque d'oublier que d'autres versions de cette fonction demandent probablement les mêmes modifications.

Nous le concevons donc, **la redondance du code est à proscrire** [1]. Comment dans ces conditions créer une seule fonction Max qui permette de définir cette fonction pour des entiers, mais aussi pour des double, des uint, etc... ? Le C++ propose un mécanisme pour définir en une fois le code devant s'appliquer, quel que soit le type des arguments. Observons la syntaxe suivante :

```
1   template<typename T>
2   T Max(T array[], int length)
3   {
4       T vmax = array[0];
5       for (int i = 1; i < length; i++)
6           if (array[i] > vmax)
7               vmax = array[i];
8       return vmax;
9   }
```

Dans cet exemple, la fonction Max devient paramétrée par un type abstrait T. Celui-ci est utilisé dans notre exemple à la fois pour définir le type du premier argument de la fonction, mais également pour définir son type de retour. Le préfixe template<typename T> indique au compilateur que le code qui suit sera paramétré par un type T. De manière équivalente, le mot clef *typename* peut être remplacé par *class*.

Lorsque dans notre code, nous voulons utiliser notre fonction Max, nous pouvons le faire de la sorte :

```
1   int values[]={ 16, 8, 3, 2, 11 };
2
3   cout << Max<int>(values, 5);
```

Lorsque le compilateur va lire l'appel à Max(values, 5), il va détecter qu'il s'agit d'utiliser la fonction templatée Max dans le cas où T = int. Le compilateur va alors générer le code correspondant et l'inclure dans la compilation. Bien

1. La règle 1 du développement pourrait être : "ne faites pas de copier/coller de code au sein d'un projet."

évidemment, le compilateur ne génèrera la fonction Max que pour les types T pour lesquels il est fait appel quelque part à la fonction Max utilisée pour le type T : si nulle part dans notre code nous ne cherchons à déterminer le max d'un tableau de double, le code spécifique pour la fonction Max en le type double ne sera pas généré.

Le fait d'utiliser une fonction templatée en un type spécifique est appelé *spécialisation*. Pouvons-nous spécialiser la fonction Max en n'importe quel type ? L'approche du C++ sur la question est une approche optimiste (à la différence du C# par exemple) : par défaut, tout type est accepté. C'est uniquement à la compilation que le compilateur va tenter de générer le code nécessaire pour chacun des types en lesquels la fonction templatée a été spécialisée. Si notre fonction est spécialisée en un type T1 pour lequel l'opérateur $<$ n'est pas défini, alors le compilateur échouera dans la génération du code spécialisé.

1.1.1 Templates et macro

Si nous reprenons la fonction templatée précédente appliquée à deux valeurs plutôt qu'à un tableau, nous pouvons produire le code suivant :

```
1 template<typename T>
2 const T & Max( const T & a, const T & b )
3 {
4     return a > b ? a : b;
5 }
```

Cet exemple ressemble beaucoup avec la macro correspondante, comme détaillée dans le chapitre sur la compilation :

```
1 #define MAX(a, b) (((a) > (b)) ? (a) : (b))
```

En effet, dans chaque cas, nous avons une implémentation de la fonction max qui peut s'adapter à tous types d'objets. Le templating est la manière propre d'écrire des macros. Là où la macro est une simple substitution syntaxique, avec toutes les erreurs qui en découlent (**??**), le templating génère de réelles fonctions et permet donc d'obtenir de manière fiable le résultat.

Dans la comparaison qui vient d'être faite, il faut noter cependant qu'un avantage majeur de la macro par rapport à son homologue templatée et l'absence d'appel de fonction : puisque la macro ne crée pas de véritable fonction, il n'y a pas d'appel de fonction et donc pas de coût d'appel de fonction. Il est possible d'éviter ce coût également dans le cas d'une fonction templatée, en l'inlinant :

```
1 template<typename T>
2 inline const T & Max( const T & a, const T & b )
3 {
4     return a > b ? a : b;
5 }
```

1.1.2 Fonctions membres templatées

Il est également possible de créer une fonction templatée au sein d'une classe non templatée. L'exemple suivant décrit une classe disposant d'une fonction membre templatée permettant d'afficher différente objets.

```
class SomeClass
{
    public SomeClass();
    public ~SomeClass();

    template<typename T>
    static void Display( const T & t )
    {
        cout << t;
    }
}

int main()
{
    SomeClass.Display<int>(2);
    SomeClass.Display<string>("Hello World");
    SomeClass.Display<double>(3.14);
}
```

1.1.3 Inférence automatique de type de spécialisation

Lorsque le compilateur est capable d'inférer le type en lequel est spécialisée une fonction templatée, il est superflu de spécifier explicitement en quel type la fonction est spécialisée. Dans le cas du code précédent par exemple, nous pourrions écrire :

```
int main()
{
    SomeClass.Display(2);
    SomeClass.Display("Hello World");
    SomeClass.Display(3.14);
}
```

Il est cependant des cas où une telle inférence n'est pas possible, notamment dans le cas d'ambiguïté que le compilateur ne peut pas lever lui-même. Ainsi, la fonction suivante doit être spécifiée explicitement :

```
template <typename T>
T Sum( T s1, T s2 )
{
    return s1 + s2;
}

int main()
{
    int s2 = 1;
    double s1 = 3.2;

    Sum( s1, s2 );
```

```
13      Sum<double>( s1, s2 );
14  }
```

1.1.4 Multi-templating

Il est possible de paramétrer une fonction par plusieurs arguments ; en voici un exemple :

```
1  template<typename T, typename U>
2  public T Add (const T& t, const U& u)
3  {
4      return t+u;
5  }
```

1.2 Templating par un type, le cas des classes

De la même manière que nous avons défini le paramétrage d'une fonction par un type générique T, nous pouvons paramétrer une classe par un type générique. Prenons directement un exemple parlant. Dans le chapitre sur la gestion de la mémoire, nous avons introduit une classe NaiveVector, appliquant le principe RAII, afin de déléguer la gestion de la mémoire associée à un tableau de doubles à une classe dédiée. Si nous voulons manipuler des tableaux d'entiers, nous devrions réécrire cette classe dans le cas des entiers, et la réécrire encore à chaque fois que nous manions un tableau d'un type distinct. Le templating se présente donc très bien à cette classe.

```
template <typename T>
class Vector<T>
{
  Vector<T>::Vector<T>(int size):m_size(size)
  {
    data = new T[size];
  }

  Vector<T>::~Vector<T>()
  {
    delete[] data;
  }

  Vector<T>::Vector<T>(const Vector<T> &v)
  {
    m_size = v.m_size;

    if (data != 0)
    {
      delete[] data;
    }

    data= new T[m_size];

    for(int i = 0; i < m_size; i++)
      data[ i ] = v.data[ i ];

  }

  Vector<T>& Vector<T>::operator=(const Vector<T> &v)
  {
    if (this == &v)
      return (*this);

    m_size = v.m_size;
    if (data != 0)
      delete[] data;

    data= new T[m_size];

    for(int i = 0; i < m_size; i++)
      data[ i ] = v.data[ i ];

    return (*this);
  }

  T & Vector<T>::operator[](int i)
  {
    return data[ i ];
  }

  int Vector<T>::getSize()
  {
    return m_size;
  }

private:
  unsigned int m_size;
  T* data;
};
```

Listing 1.1 – vectorTemplate.h

En dépit de son aspect ultra-générique, le code présenté ci-dessus ne fonctionnera que pour les classes T disposant d'un constructeur ne prenant pas d'argument. En effet dans le cas contraire, il est impossible d'instancier un tableau de type T par la commande data = new T[size].

Au delà de cette remarque restrictive, nous observons que les templates sont un excellent moyen de créer des classes containers comme des tableaux ou des listes. Une bibliothèque classique, la STL (Standard Template Library), permet ainsi de gérer simplement des ensembles d'objets de manière générique. Nous renvoyons le lecteur au chapitre sur les containers pour une discussion plus approfondie sur le sujet.

1.3 Templates et compilation

Un voile pudique est bien souvent jeté par les manuels d'introduction au C++ sur la compilation des templates. Ceux-ci répondant à des contraintes bien particulières (puisqu'il ne s'agit pas de code mais de "méta-code"), il n'est pas possible par défaut de déclarer une fonction template dans un fichier .h puis de la définir dans un fichier .cpp. Par défaut, il vous faudra donc mélanger déclaration et définition dans un même fichier .h sous peine de vous exposer à des erreurs à l'édition des liens. Comme il est expliqué dans l'article d'Aurélien Regat-Barrel sur le site cpp.developpez.com, il existe néanmoins une astuce permettant de contourner le problème.

Cette astuce consiste à stocker la déclaration de votre classe templatée dans un fichier .h, de stocker votre définition dans un fichier texte (avec une extension différente de .cpp, comme .tpp par exemple), et d'inclure grâce à l'instruction #include le fichier .tpp à la fin du fichier header. Ainsi, nous obtenons par exemple quelquechose de la forme :

```
1  //exemple.h
2
3  #ifndef EXEMPLE_H
4  #define EXEMPLE_H
5
6  template <typename T>
7  class Exemple
8  {
9  public:
10     Exemple();
11 };
12
13 #include "exemple.tpp"
14 #endif
15
16 //exemple.tpp
17
18 template <typename T>
```

```
19 Exemple<T>::Exemple()
20 {
21 }
```

1.4 Templates et spécialisation

Il existe certains cas où nous voudrions faire des exceptions à la généricité, c'est à dire que pour certains types bien particuliers, une fonction templatée ait un comportement particulier, qui diffère du comportement général déjà défini. Prenons le cas de l'exponentiation de 2. Si y est un réel (double ou float), le calcul de 2^y demande de réécrire la formule en $e^{y.ln(2)}$ afin de l'évaluer. Nous pourrions donc écrire :

```
1
2 template<typename T>
3 double TwoPow(T y)
4 {
5     return exp(y*ln(2);
6 }
```

Cette fonction fonctionnerait également si elle était spécifiée en deux entiers. Cependant, dans le cas où y est entier, il n'est pas nécessaire de passer par cette formule, il suffit alors de multiplier 2 par lui même y fois. Bien que la formule précédente soit exacte dans le cas où y est entier, elle entrainerait donc des calculs indûment longs. Pour améliorer cette situation, nous voudrions dire au compilateur : compile la fonction précédente pour tous les types nécessaires, SAUF dans le cas où y est entier, auquel cas contente toi de calculer directement la valeur de l'exponentiation. En C++, il est possible de préciser/redéfinir une spécialisation spécifique.

```
1
2
3 template <>
4 double TwoPow<int>( int i )
5 {
6     double q= (i >= 0) ? 2 : 0.5;
7     int iAbs = abs(i);
8     double r=1;
9
10    for (int j = 0 ; j < iAbs;j++)
11        r*=q;
12
13    return r;
14 }
```

1.4.1 Spécialisation partielle

Les templates peuvent être partiellement spécialisés, et la classe obtenue est alors encore un template. Cette spécialisation partielle intervient principalement

dans le cas dans le cas d'un template paramétré par plusieurs types, pour lesquels seuls certains de ces types sont spécialisés, le résultat étant un template paramétré dans les types restants. Exemple :

```
template<typename T, typename U>
public double Pow(T x, U y)
{
    return exp(y*ln(x));
}

template<typename T>
public double Pow<T,int>(T x, int y)
{
    double q= (y >= 0) ? x : ((double)1)/x;

    int yAbs = abs(y);
    double r=1;

    for (int j = 0 ; j < yAbs;j++)
        r*=q;

    return r;
}
```

1.5 Templating par des entiers

Il est également possible de paramétrer une fonction par autre chose qu'un type. Notamment, il est possible de paramétrer une fonction par un entier. Ces mécanismes ne seront pas détaillés cette année, mais ils sont massivement utilisés dans de nombreuses librairies professionnelles, car ils permettent des optimisations très fines, notamment via le Template Meta-Programming. Les bonnes librairies de calcul scientifique en C++ par exemple reposent toutes massivement sur ce genre d'optimisation. Nous renvoyons le lecteur intéressé par exemple à [?].