Templates

Xavier Juvigny

19 mars 2020

Les templates sont une puissante caractéristique du C++ qui vous permet d'écrire des programmes génériques, c'est à dire dont une partie du code est générée par le compilateur C++ lui-même ! En des termes simples, vous pouvez écrire des patrons (comme pour la couture...) de fonctions ou de classes qui vont permettre de générer des fonctions (ou des classes) avec divers types de données.

Les templates sont souvent utilisés dans le but de reutiliser du code pour divers types de données et pour rendre un programme plus flexible.

Le concept de templates peut être utilisé de deux façons différentes :

- Pour créer des patrons de fonctions : on parle de **généricité fonctionnelle** qui permet de créer une fonction pour un ensemble de types de valeurs. Par exemple, échanger deux valeurs `a` et `b` s'écrit de la même manière quelques soient le type de `a` et `b`.
- Pour créer des patrons de classes : on parle de **généricité structurelle** qui permet de créer des structures génériques qui ne présupposent pas le type des données gérées en interne de la structure. Par exemple, il peut être utile de créer une liste chaînée générique qui permettra par la suite aussi bien de gérer une liste chaînée de réels qu'une liste chaînée de chaînes de caractères. La gestion de ces deux listes est complètement identique, seul le type des valeurs gérées par les deux listes diffère !

Prenons par exemple la fonction `max` qui compare deux valeurs et retourne la plus grande valeur. Clairement, cette fonction s'écrit rigoureusement de la même façon pour un très grand nombre de types :

```
0  int max(int a, int b)
   {
2      return a > b ? a : b;
   }
4
   float max(float a, float b)
6  {
       return a > b ? a : b;
8  }

10 double max(double a, double b)
   {
12     return a > b ? a : b;
   }
```

```
...
```

Le corps de chacun de ces fonctions (surchargée grâce au C++), s'écrit exactement de la même manière.

En C, pour écrire une fonction `max` générique, on utilise les macros :

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

Cela marche plutôt bien pour des appels simples :

```
int a = MAX(3,5); // a vaut 5
double b = MAX(0.5*a, 7./a); // b vaut 2.5
```

Cette fonction marchera pour tout type d'argument du moment qu'on puisse comparer la valeur a avec la valeur b. Cependant, l'utilisation des macros est dangereux. En effet, quelles valeurs auront `a` et `c` après avoir exécuté les lignes suivantes ? (Exercice de C)

```
int c = MAX(a++, 4);
```

On s'attendrait à ce que `a` vaille 6 et `c` vaille 5. Néanmoins, si on affiche les valeurs de `a` et `c`, on trouvera que `a` vaut 7 et `c` vaut 6.

Pourquoi ?

En fait, dans une macro, on remplace le paramètre `a` par la première expression donnée dans l'appel à la macro, et le paramètre `b` par la seconde expression, tout cela dans la macro elle-même, si bien que

```
int c = MAX(a++,4);
```

est équivalent à écrire :

```
int c = ((a++) > (4) ? (a++) : (4));
```

ce qui revient ici à incrémenter `a` une première fois après comparaison avec quatre, et puisque `a` valait cinq en entrée (et donc supérieur à 4 !), on retourne sa valeur et on l'incrémente, ce qui explique les deux valeurs trouvées.

C++ nous permet d'écrire des fonctions génériques (ou des structures) sans avoir recours aux macros, d'une manière bien plus sûre et surtout bien plus puissante.

1 Les patrons de fonction

Un patron de fonction fonctionne d'une manière similaire à une fonction *usuelle*, avec une différence clef : une fonction usuelle ne peut être appelée qu'avec un type et les types convertibles en ce type tandis qu'un patron de fonction va pouvoir être appelée avec divers types non convertibles entre eux.

Normalement, avec les fonctions usuelles, vous allez devoir, pour divers types non convertibles entre eux, créer autant de fonction avec le même nom que de types de variables non convertibles devant l'appeler. Avec les templates, vous allez pouvoir créer un seul patron de fonction que le compilateur utilisera pour générer autant de fonctions que de types de paramètres différents utilisant cette fonction.

1.1 Déclaration d'un patron de fonction

Un patron de fonction commence avec le mot clef `template` suivi entre les symboles < et > par les paramètres "templates" puis par le reste de la déclaration de la fonction, comme pour une fonction usuelle.

```
template<typename T>
T une_fonction(T args)
{
    ... .. ...
}
```

Dans le code ci–dessus, `T` est un argument template qui accepte divers types de données (int, float, etc.) et `typename` est un mot clef.

Quand un argument est passé à cette fonction, le compilateur va générer une version de `une_fonction` pour le type de la donnée passée en argument.

Exemple : trouver l'élément le plus grand dans une collection d'éléments

```
#include <iostream>

// Patron de fonction
template<typename K> max_val(const K& a, const K& b)
{
    return a > b ? a : b;
}

// Programme principal :
int main()
{
    std::cout << "max(1,3) = " << max_val(1,3) << std::endl;
    std::cout << "max(3.14,2.54) = " << max_val(3.14, 2.54) << std::
    endl;
    return EXIT_SUCCESS;
}
```

Dans le programme au dessus, une fonction template `max_val` est définie et accepte deux arguments `a` et `b` dont le type de donnée est `K`. `K` ici est un argument template, il signifie donc que l'argument peut être de n'importe quel type du moment qu'on puisse comparer deux valeurs de ce type.

Dans la fonction principale, les deux entiers et les deux réels sont passés à la fonction template comme à une fonction normale. Durant la phase de compilation, le compilateur détecte qu'on a appelé la fonction `max_val` avec deux entiers et avec deux réels. Il générera donc durant la compilation deux fonctions `max_val`, l'une demandant deux entiers en argument et l'autre deux réels.

Rajoutons à notre test la ligne suivante :

```
0     std::cout << "max(1,3.5) = " << max_val(1, 3.5) << std::endl;
```

Lors de la compilation du source, une surprise nous attend : le compilateur nous gènère une erreur, nous avertissant qu'il n'a pas trouvé de fonction qui correspond aux types passés dans l'appel. Pourquoi ?

Lors de l'appel de la fonction `max_val` avec les valeurs 1 et 3.5, on passe en argument à la fonction un entier et un réel double précision. Or, dans notre fonction, le compilateur s'attend à avoir pour `val1` et `val2` deux valeurs de même type (puisqu'ils sont tous les deux de type `T`).

Il faut donc permettre à notre fonction d'accepter deux valeurs avec deux types différents, du moment que ces deux valeurs puissent être comparables.

Ainsi, on modifie notre code pour généraliser notre fonction :

```
0  #include <iostream>
2  template<typename T1, typename T2> auto max_val( const T1& val1, const
       T2& val2)
   {
4      return val1 > val2 ? val1 : val2;
   }
6
   int main()
8  {
       std::cout << "max_val(1,3) = " << max_val(1,3) << std::endl;
10     std::cout << "max_val(1.5,4.5) = " << max_val(1.5,4.5) << std::endl
       ;
       std::cout << "max_val(1,3.5) = " << max_val(1, 3.5) << std::endl;
12     return EXIT_SUCCESS;
   }
```

Cette fois ci, le code compile et nous retourne la valeur attendue.

Rajoutons maintenant la ligne de test suivante :

```
0     std::cout <<  max_val("tintin", "milou") << std::endl;
```

Cela dépend des compilateurs, mais la valeur attendue ne sera pas forcément "tintin" ! En fait, "tintin" et "milou" sont deux chaînes de caractères en C, pris comme des tableaux de caractères. Par défaut, le C, et donc le C++, pour un tableau, compare la valeur des pointeurs, et si le pointeur de "milou" est plus grand que le pointeur de "tintin", notre fonction retournera "milou", ce qui n'est probablement pas ce qu'on voulait que la fonction fasse !

Il va falloir donc "spécialiser" la fonction pour les chaînes de caractère de type C pour préciser que dans ce cas, on effectuera bien une comparaison de chaîne de caractère. Dans un premier temps, nous allons donc modifier notre code de la façon suivante :

```
#include <iostream>

template<typename T1, typename T2> auto max_val( const T1& val1, const
    T2& val2)
{
    return val1 > val2 ? val1 : val2;
}

const char* max_val( const char* s1, const char* s2)
{
    if (strcmp(s1,s2) > 0) return s1; else return s2;
}

int main()
{
    std::cout << "max_val(1,3) = " << max_val(1,3) << std::endl;
    std::cout << "max_val(1.5,4.5) = " << max_val(1.5,4.5) << std::endl
    ;
    std::cout << "max_val(1,3.5) = " << max_val(1, 3.5) << std::endl;
    std::cout << max_val("tintin", "milou") << std::endl; // Retourne
    la bonne valeur !
    return EXIT_SUCCESS;
}
```

Enfin, remarquons qu'à partir de C++ 20, il est possible d'utiliser le mot clef `auto` (avec les conceptes qu'on ne verra pas ici) pour définir une fonction patron :

```
#include <complex>
#include <iostream>
auto max_val( const auto& val1, const auto& val2)
{
    return val1 > val2 ? val1 : val2;
}

const char* max_val( const char* s1, const char* s2)
{
    if (strcmp(s1,s2) > 0) return s1; else return s2;
}

int main()
{
```

```
    std::cout << "max_val(1,3) = " << max_val(1,3) << std::endl;
    std::cout << "max_val(1.5,4.5) = " << max_val(1.5,4.5) << std::endl
    ;
    std::cout << "max_val(1,3.5) = " << max_val(1, 3.5) << std::endl;
    std::cout <<  max_val("tintin", "milou");
    return EXIT_SUCCESS;
}
```

ce qui permet d'alléger la lecture du code. On parle alors de **spécialisation template**. C++ n'instanciera pas une fonction template si une version non template de cette fonction existe déjà. Il utilisera plutôt la fonction existante.

Enfin, signalons qu'il est tout à fait possible de définir une méthode d'une classe comme template, à la seule condition que cette fonction ne soit pas virtuelle (à cause de la table des fonctions virtuelles dont la taille doit être connue à la compilation de la classe).

1.2 Les paramètres templates

En fait, les paramètres templates ne se limitent pas à un type générique. Un paramètre template peut-être :

- Un **type générique** : c'est un paramètre précédé du mot clef typename (recommandé) ou (class)
- Un **type intégral** : c'est à dire un entier, un booléen, mais aussi un pointeur, une référence, etc. Tout ce qui est énumérable.

1.3 Le cas du pointeur nul

Le pointeur nul est un pointeur un peu particuliers, qui avant le C++ 11 était simplement considéré comme un entier long de valeur nulle. Considérons le code suivant :

```
#include <iostream>
template<typename T> void display_pointer( const T* pt)
{
    std::cout << "Adresse du pointeur : " << pt << " avec des éléments
    de taille " << sizeof(T) << std::endl;
}

int main()
{
    double x = 0; int i = 3;
    display_pointer(&x)  ; display_pointer(&i);
    display_pointer(NULL); display_pointer(nullptr);
    return EXIT_SUCCESS;
}
```

Si on essaie de le compiler, on s'aperçoit que le compilateur ne trouvera pas de fonctions appelables avec les paramètres `NULL` et `nullptr`. En effet, comme

dit plus haut, `NULL` est un entier, et non un pointeur comme la fonction template s'y attend et `nullptr` possède sa propre structure `std::nullptr_t` (`NULL` peut être converti implicitement en type `std::nullptr_t` sur certains compilateurs , mais pas tous !).

Pour pouvoir gérer également les pointeurs nuls, il va donc falloir spécialiser la fonction template :

```
#include <iostream>
template<typename T> void display_pointer( const T* pt)
{
    std::cout << "Adresse du pointeur : " << pt << " avec des éléments
    de taille " << sizeof(T) << std::endl;
}

void display_pointer( std::nullptr_t pt )
{
    std::cout << "Pointeur nul !" << std::endl;
}

int main()
{
    double x = 0; int i = 3;
    display_pointer(&x)  ; display_pointer(&i);
    display_pointer(NULL); display_pointer(nullptr);
    return EXIT_SUCCESS;
}
```

Puisqu'un paramètre template peut être un paramètre de type entier, on peut également spécialiser un template pour une valeur particulière de ce paramètre.

Par exemple, on peut définir la fonction factorielle à partir d'un template :

```
#include <iostream>
template<long n> constexpr long fact() {return n * fact<n-1>(); }
// Spécialisation template pour la valeur 0 :
template<> constexpr long fact<0>() { return 1L; }

int main()
{
    std::cout << fact<10>() << std::endl;
    return EXIT_SUCCESS;
}
```

L'avantage de définir ici la factorielle comme une fonction template est que l'évaluation de 10! se fait à la compilation et non durant l'exécution (si on rajoute au minimum un niveau d'optimisation à la compilation (-O)).

Le pouvoir du template de pouvoir évaluer des fonctions à la compilation à donner naissance à toute une classe de puissantes techniques en C++ qui ont au final détourné le but premier des templates de générer des classes et des fonctions. On verra un peu plus loin certaines de ces techniques.

Supposons maintenant qu'on veuille écrire une fonction évaluant la dérivée direction d'une fonction sur un corps `K` (réel, complexe, etc.). Une mise en œuvre possible de cette fonction est la suivante :

```
#include <iostream>
#include <complex>
using namespace std::literals::complex_literals;
#include <cmath>

/** Le premier paramètre template est le type de la fonction à évaluer.
     Le deuxième est le corps où la fonction prend ses valeurs
*/
template<typename Function, typename K> auto df_s_dh( const Function& f
    , const K& h, const K& x,
                                                      const decltype((
    std::abs(h)))& eps = 1.E-6 )
{
    auto norm_h = std::abs(h); // On devrait lancer une exception sir
    norm_h est plus petit qu'une certaine valeur. Possible mais pas
    fait ici.
    K dh = eps * h / norm_h;
    return K(1./eps) * (f(x+dh) - f(x));
}

/** On spécialise la dérivée directionnelle pour les réelles en la
    transformant en simple dérivée :
*/
template<typename Function> double df_s_dh(const Function& f, const
    double& h, const double& x)
{
    return (1./h)*(f(x+h)-f(x));
}

double f(double x)
{
    return std::sin(x*x);
}

std::complex<double> g(const std::complex<double>& z)
{
    return z * std::exp(z);
}

int main()
{
    double x = 3.;
    std::complex<double> z = 0.5 + 1.i;
    auto dsqrt   = df_s_dh((double(*)(double))std::sqrt, 1.E-6, x);
    auto df = df_s_dh(f, 1.E-6, x);
    auto dg = df_s_dh(g, 1. + 1.i, z);
    auto dh = df_s_dh([](const std::complex<double>& z){ return z*z; },
     1. + 1.i, 0. + 0.5i);

    std::cout << "Pour x = 3, on a :" << std::endl;
    std::cout << "d sqrt(x) = " << dsqrt << std::endl;
    std::cout << u8"d sin(x²) = " << df << std::endl;
    std::cout << "d z.e^z /d(1+i) (0.5+i) = " << dg << std::endl;
```

```
    std::cout << "d z*z / d(1+i) (0.5 i) = " << dh << std::endl;
    return EXIT_SUCCESS;
}
```

Ce programme, un peu long appelle à beaucoup de commentaires.

— Le type de la fonction est passée comme argument template. En effet, plusieurs signatures sont possibles pour les fonctions passées en argument : elles peuvent avoir un argumetn par valeur ou par référence constante, ce qui les différencie quant à leur type.

— Le fait de passer le type de la fonction en argument template permet également de pouvoir passer en argument un objet possédant l'opérateur `()` qui permet d'évaluer l'objet comme une fonction (on peut penser par exemple à des fonctions paramétriques dont les paramètres sont stockés dans un objet)

— On peut également passer des lambdas fonctions, c'est à dire des fonctions anonymes dont le type est unique pour chaque lambda fonction ! Il serait impossible d'utiliser les fonctions lambdas si on n'avait pas passer le type de la fonction dans les paramètres templates !

— Vous pouvez remarquer pour la fonction `std:sqrt` que j'ai été obligé de repréciser en conversissant que `std::sqrt` est une fonction prenant un double comme argument et qui retourne un double. En effet, il existe plusieurs versions de `std::sqrt` qui prennent un float, un double ou un long double. Passer uniquement `std::sqrt` comme fonction ne permet pas au compilateur de déduire la valeur de l'argument template puisque plusieurs versions de cette fonction existe.

Plutôt que d'évaluer une expression à partir d'une fonction template comme pour la factorielle plus haut, on peut aussi utiliser à partir de C++ 14 des templates sur les expressions constantes. Ainsi le code précédent peut être reécrit comme suit :

```
#include <iostream>

template<typename I, long n> constexpr I factoriel = I(n) * factoriel<I
    ,n-1>;
template<typename I> constexpr I factoriel<I,0> = I(1);

int main()
{
    std::cout << "10! = " << factoriel<unsigned long,10> << std::endl;
    std::cout << "20! = " << factoriel<double,20> << std::endl;
    return EXIT_SUCCESS;
}
```

Plusieurs avantage à cette reécriture de la fonction factorielle :

— On est assuré même sans optimisation que le compilateur évalue 10 ! et 20 ! à la compilation

— Le fait d'avoir passé en argument template le type de la valeur retourner permet pour des valeurs importantes passées à la factorielle de pouvoir utiliser les réels double précision au lieu d'entiers.
— On a utiliser la spécialisation partielle pour le type de retour de la fonction factorielle.

Il faut en fait bien comprendre le mécanisme des templates en C++ pour arriver à maîtriser cet outil puissant mais complexe !

Le principe sur lequel se basent tous les compilateurs se nomme SFINAE (Substitution Failure Is Not An Error).

Le compilateur va chercher tout d'abord si il existe des fonctions (ou des structures) non templates dont les types des paramètres correspondent aux types des arguments passés à l'appel (en tant que paramètres templates ou pour des fonctions, en arguments de la fonction) et utilise cette fonction/classe si il le trouve.

Dans le cas où le compilateur ne trouve pas de version adéquat, il va rechercher si il existe des fonctions/classes dont la spécialisation partielle peut être utilisée sans que la compilation échoue. Si c'est le cas, le compilateur utilise cette fonction ou certte classe spécialisée partiellement (à condition qu'elle soit unique, sinon il enverra un message d'erreur).

Enfin, en dernier lieu, le compilation cherchera si il existe une fonction ou une classe template (il peut y avoir plusieurs versions templates pour une même fonction ou une structure) avec laquelle la compilation n'échoue pas. Si cela se traduit par un échec, le compilateur renverra une erreur (avec pour certaines versions de compilateur, la liste de toutes les tentatives échouées, ce qui peut donner un message d'erreur très indigeste !).

Ce comportement du compilateur lors de l'utilisation des templates est largement utilisé par les programmeurs C++ au travers une classe de technique qu'on appelle SFINAE (Substitution Failure Is Not An Error). Cela permet en autre de faire de l'introspection avec C++ lors de la phase de compilation (hors cadre de ce cours, c'est du C++ avancé).

2 Généricité structurelle

Comme pour les fonctions templates, les templates sont utiles pour définir une structure/une union/un type qui est indépendant de type de donnée. Les classes templates sont par exemple utiles pour définir des types comme des listes chaînées, des arbres n-aires, des piles, des queues, des tableaux statiques ou dynamique. C'est d'ailleur en grande partie ces structures qui sont définie en template dans la bibliothèque standard du C++, la STL (Standard Template Library !).

Un exemple simple (et simplifié) pourrait être la structure `pair` qui peut se programmer comme suit :

```
0 #include <iostream>
  #include <string>
2
```

```
template<typename T1, typename T2>
struct pair
{
    T1 first;
    T2 second;

    pair( const T1& p1, const T2& p2 ) : first(p1), second(p2)
    {}
};

template<class T1, class T2> pair<T1,T2> make_pair( const T1& p1, const
     T2& p2)
{
    return {p1,p2};
}

int main()
{
    auto pair1 = make_pair(3, std::string("toto"));
    auto pair2 = make_pair(3, 0.1415);
    auto pair3 = pair<double, int>(4.5,2);
    std::cout << "pair 1 : " << pair1.first << "," << pair1.second <<
    std::endl;
    std::cout << "pair 2 : " << pair2.first << "," << pair2.second <<
    std::endl;
    std::cout << "pair 3 : " << pair3.first << "," << pair3.second <<
    std::endl;
    return 0;
}
```

Notons la fonction `make_pair` qui nous permet de créer des pairs de "choses" sans à avoir à le préciser dans les paramètres templates comme cela est fait pour la troisième paire. En effet, au contraire des fonctions, jusqu'au C++ 17 du moins, le compilateur était incapable de deviner les paramètres templates par rapport aux paramètres passés à un constructeur de la classe. Remarquons par ailleurs que si on veut utiliser le constructeur par défaut (non défini dans notre exemple, mais bon...), il faudrait écrire par exemple la ligne suivante :

```
pair<double, std::complex<double>> zpair;
```

Comme pour les fonctions, il est possible d'avoir des paramètres par défaut template. Par exemple, en reprennant notre exemple, on peut vouloir que le second élément de la paire soit une chaîne de caractère :

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

template<typename T1, typename T2 = std::string>
struct pair
{
    T1 first;
    T2 second;
```

```
8
    pair( const T1& p1, const T2& p2 ) : first(p1), second(p2)
10  {}
};
12
int main()
14 {
    auto pair1 = pair<double>(4.5,std::string("toto")); // Le second
    paramètre est par défaut une chaîne de caractère...
16  std::cout << "pair 1 : " << pair1.first << "," << pair1.second <<
    std::endl;
    return 0;
18 }
```

Les méthodes déclarées au sein d'une classe template mais définies en dehors doivent elles-mêmes être redéfinies template :

```
0 template<typename K> class A
{
2     ...
    K func(const K& k ) const;
4 };
...
6 template<typename K> K A<K>::func(const K& k ) const
{
8    ...
}
```

Attention cependant, lors de l'instanciation d'une classe lors de la compilation, le compilateur doit avoir le code complet de la classe à portée de main (c'est à dire les déclarations mais aussi les définitions). En général, pour une classe template, tout est mis au sein d'un même ficher d'entête.

Un exemple concret d'une telle classe template est donné dans le répertoire `Exemple/vecteur_template`.