Je pratique le C++

Partie 3/5.

Nous vous proposons une série d'articles sur la pratique de C++ pour que vous puissiez tous vous y mettre. Ce mois-ci on aborde les templates C++. Les templates représentent la partie programmation générique du langage C++.

Level 400

Christophe PICHAUD | .NET Rangers by Sogeti
Consultant sur les technologies Microsoft
christophepichaud@hotmail.com | www.windowscpp.net

La programmation orientée objet (OOP) et la programmation générique travaillent avec des types qui ne sont pas connus au moment où le programme est écrit. La distinction entre les deux est que OOP travaille avec des types qui ne sont connus qu'au runtime, tandis que la programmation générique utilise des types qui sont connus à la compilation.

Les containers, les itérateurs et les algorithmes sont des exemples de programmation générique. Quand on écrit un programme générique, le code est indépendant des types qu'il manipule. Quand on utilise un programme générique, nous fournissons les types ou les valeurs sur l'instance du programme qui va tourner. Par exemple, la librairie standard fournit une simple définition générique de chaque container, comme vector. Nous pouvons utiliser la définition générique pour définir plusieurs types de vector, chacun étant différent des autres, suivant le type qu'il contient. Les templates sont les fondations de la programmation générique en C++. Nous pouvons les utiliser sans comprendre comment ils sont définis. Un template est une « formule » pour créer des classes ou des fonctions. Lorsque nous utilisons un type générique comme vector, ou une fonction générique comme find, on fournit les informations de type nécessaire à l'exécution de cette classe ou fonction à la déclaration. Exemple : vector<string> v.

Concepts et programmation générique

Pourquoi les templates sont-ils fait ? Quelles techniques de programmation sont mises en œuvre quand on utilise les templates ? Les templates offrent :

- La possibilité de passer des types (des valeurs et des templates) en tant qu'arguments sans perte d'information.
- La vérification de type faite à l'instanciation.
- La possibilité de passer des valeurs constantes comme arguments. Cela implique de faire du calcul à la compilation.

Les templates fournissent un mécanisme puissant de compile-time computation et de manipulation de type qui permettent d'avoir un code efficace et très compact. Rappelons-nous que les types (classes) peuvent contenir du code et des valeurs. Le premier et le plus commun scenario d'utilisation des templates est le support de la programmation générique, qui est un modèle de programmation mettant l'accent sur le design, l'implémentation et l'utilisation d'algorithmes généraux. Ici, « général » veut dire qu'un algorithme peut être conçu pour accepter un large panel de types tant qu'ils sont passés en arguments. Les templates sont (compile-time) un mécanisme de polymorphisme paramétrique.

Considérez la fonction sum :

```
template<typename Container, typename Value>
Value sum(const Container& c, Value v)
{ for ( auto x : c ) v+=x;
return v;
}
```

Elle peut être invoquée pour n'importe quelle structure de données qui supporte begin() et end() de telle manière que le range-for puisse s'exécuter. De telles structures de la librairie standard (Standard Template Library) comme vector, list et map font l'affaire. Pour aller plus loin, le type d'élément de cette structure de données est seulement limité par son utilisation : elle doit être un type auquel on peut ajouter un argument Value. Les exemples sont ints, doubles et Matrices.

On peut dire que l'algorithme sum() est générique dans 2 dimensions : le type de data structure pour stocker les éléments (le container) et le type de ses éléments. Donc, sum() requiert que le premier argument soit une sorte de container, et le second argument de template soit une sorte de nombre. De tels prérequis se nomment des concepts.

De bons et précieux concepts sont fondamentaux pour la programmation générique. Les exemples sont les entiers et les nombres à virgules flottantes (comme définis même en C classique), et plus généralement des concepts mathématiques comme les vector et les containers. Ils représentent les concepts fondamentaux pour un champ d'applications. L'identification et la formalisation à un degré nécessaire pour une programmation générique efficace peut être un challenge.

Pour une utilisation basique, considérez le concept Régulier. Un type est régulier lorsqu'il se comporte comme un int ou un vector.

Un objet de type régulier :

- Peut être construit par défaut
- Peut être copié en utilisant un constructeur ou une affectation
- Peut être comparé en utilisant == et !=
- Ne souffre pas de problème technique à l'utilisation

Une string est un autre exemple de type régulier. Comme int, string est aussi Ordonné.

Cela veut dire que 2 strings peuvent être comparées avec <, <=, >, >= avec les sémantiques appropriées. Les concepts, ce n'est pas juste une notion syntaxique, c'est fondamentalement de la sémantique.

Les fonctions templates

Imaginons une fonction compare qui fonctionne avec tous les types, comment allons-nous écrire cela :

```
template <typename T>
int compare(const T &v1, const T &v2

{
   if (v1 < v2) return -1;
   if (v2 < v1) return 1;
   return 0;
}
```

Le return 0 est juste là pour faire joli car il faut finir la fonction et retourner quelque chose... C'est une simple fonction qui prend deux types T en paramètre. Remarquez l'élégance du style. Dans une définition de template, la liste des paramètres de template ne peut être vide.

Une définition de template commence avec le mot-clé template suivi d'une liste de paramètres de template qui sont séparés par des virgules et qui sont entre les token < et >.

Instanciation d'une fonction template

Quand on appelle une fonction template, le compilateur utilise les arguments de l'appel pour déduire les arguments de template pour nous. Lors

de l'appel à compare, le compilateur utilise le type des arguments pour déterminer quel type associer au paramètre de template T.

```
cout << compare(1, 0) << endl;      // T est int
vector<int> vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare(vec1, vec2) << endl; // T est vector<int>
```

Pour le premier appel, le compilateur va écrire et compiler une version de compare avec T remplacé par int :

```
int compare(const int &v1, const int &v2)
{
  if (v1 < v2) return -1;
  if (v2 < v1) return 1;
  return 0;
}
```

Pour le second appel, la fonction sera générée avec des vector<int>.

Les paramètres de type template

Il est possible d'utiliser le paramètre de type template comme les arguments de la fonction. Exemple :

```
template <typename T> T foo(T* p)
{
  T tmp = *p;
  // ...
  return tmp;
}
```

Chaque paramètre de type doit être précédé du mot clé class ou typename.

```
template <typename T, U> T calc(const T&, const U&);
template <typename T, class U> calc (const T&, const U&);
```

La compilation des templates

Quand le compilateur voit la définition d'un template, il ne génère pas de code. Il génère du code seulement quand on instancie une instance spécifique d'un template.

Le fait que le code soit généré seulement quand on utilise le template (et non quand on le définit) affecte la manière avec laquelle nous organisons le source code et la manière avec laquelle les erreurs sont détectées. Il en va aussi de la taille de l'exe ou de la librairie lib ou dll.

Quand on appelle une fonction, le compilateur a besoin de voir seulement la déclaration de la fonction. Quand on utilise un objet de type de class, la définition de classe doit être disponible mais la définition des fonctions membres n'a pas besoin d'être présente.

Donc, on met les définitions de classes et les déclarations de fonctions dans des fichiers d'entêtes (.h), et les définitions des fonctions membres de classes dans le source code (.cpp).

Les templates sont différents. Pour générer une instanciation, le compilateur a besoin d'avoir le code qui définit une fonction template ou une fonction membre de classe template.

Donc, les entêtes (.h) pour les templates contiennent leurs définitions et leurs déclarations et leurs fonctions membres.

Les erreurs de compilation des templates

La détection des erreurs sur les template peut être parfois un véritable parcours du combattant car le compilateur vérifie plusieurs choses lors de l'instanciation du template. Une des erreurs les plus fréquentes est celle rencontrée sur les types. Par exemple : un template requiert une opération cout <<. La compilation ne détecte pas d'erreur tant que l'instanciation sur un type n'est pas effectuée. Si le type ne supporte pas l'opérateur << pour iostream cout sur l'objet passé dans le paramètre, l'erreur est immédiatement détectée.

Exemple :

```
template <typename T> void Echo(const T & t)
{
        cout << t << endl;
}

void Template1()
{
        string t = "My Lisa";
        Echo(t); // OK
        Echo(9); // OK

        vector<string> girls = { "edith", "lisa", "audrey" };
        Echo(girls); // KO, binary << no operator found in vector<T>...
}
```

Erreur de compilation : binary '<<': no operator found which takes a right-hand operand of type 'const std::vector<std::string,std::allocator<_Ty>>' (or there is no acceptable conversion)

Dans notre exemple, il n'existe aucun opérateur << défini dans vector<T> qui puisse travailler avec la fonction cout définie dans iostream de la STL. La fonction Echo ne fait que un cout << mais ce n'est pas possible sur vector<T>. C'était possible sur string et sur int mais pas sur vector<T>.

Une bonne pratique consiste à essayer de minimiser le nombre de prérequis demandés sur le type d'argument.

C'est au designer du template de vérifier que les arguments passés au template supportent toutes les opérations que le template utilise et que ces opérations se comportent correctement dans le contexte que le template utilise.

Les templates de classes

Un template de classe sert à générer des classes. Les templates de classes diffèrent des templates de fonctions dans le fait que le compilateur ne peut pas déduire les types de paramètres du template pour un template de classe. Pour utiliser un template de classes, nous devons fournir une information additionnelle dans les token (< et >) juste après le nom du template. Les informations en extra sont la liste des arguments du template à utiliser pour ce template.

Définir un template de classe

Comme les templates de fonctions, les templates de classes commencent par le mot-clé template suivi d'une liste de paramètres du template. Voici un template pour la gestion des pointeurs. Plus besoin de faire de delete, le template s'en charge.

```
template<typename T>
class MyPtr
{
public:
  MyPtr()
  {
```

```
    m_count = 0;
    m_ptr = nullptr;
    cout << "No Pointer catched" << endl;
}

MyPtr(T* ptr) : m_ptr(ptr)
{
    m_count = 0;
    m_count++;
    cout << "Pointer catched" << endl;
}

virtual ~MyPtr()
{
    m_count--;
    if (m_count == 0)
    {
        delete m_ptr;
        cout << "Pointer deleted !" << endl;
    }
}

T& operator*(void)
{
    return *m_ptr;
}

T* operator->(void)
{
    return m_ptr;
}

MyPtr& operator=(MyPtr<T> &ptr)
{
    if ( m_ptr != nullptr)
        delete m_ptr;
    m_ptr = ptr.m_ptr;
    m_count++;
    return *this;
}

MyPtr& operator=(T* ptr)
{
    if (m_ptr != nullptr)
        delete m_ptr;
    m_ptr = ptr;
    m_count++;
    return *this;
}

private:
  T * m_ptr;
  int m_count;
};
```

Le template MyPtr possède un type de paramètre template nommé T. Nous pouvons utiliser ce paramètre n'importe où pour représenter le type que MyPtr détient. Par exemple, nous définissons le type de retour d'une opération qui fournit accès à l'élément de MyPtr comme T&.

Quand un développeur va instancier ce template, les utilisations de T seront remplacées par un argument de type template spécifique.

Instanciation du template de classe

Pour instancier le template de classe, il faut fournir explicitement un type de paramètre au template comme, par exemple, une classe CElement qui est une classe fictive pour désigner des éléments à dessiner:

```
MyPtr<CElement> pElement(new CElement(10));
pElement->Draw();
```

Il est possible d'avoir en données membre un stockage des éléments dans vector par exemple en utilisant le paramètre de type du template :

```
  std::shared_ptr<std::vector<T>> data;
```

Le type T est type comme les autres donc on peut l'utiliser comme on veut, en T, en T& en T*. Toutes les combinaisons sont possibles.

Les fonctions membres des templates de classe

Les fonctions membres peuvent être définies dans le header ou dans le corps. Si elle le sont hors du header, il faut spécifier template<T> et le nom de la classe suivi de :: et du nom de la méthode.
Exemple :
Dans le header :

```
  MyPtr& operator=(MyPtr<T> &ptr);
```

Dans le cpp :

```
template<typename T>
MyPtr<T>& MyPtr<T>::operator=(MyPtr<T> &ptr)
{
  if (m_ptr != nullptr)
      delete m_ptr;
  m_ptr = ptr.m_ptr;
  m_count++;
  return *this;
}
```

Dans le template MyPtr<T>, on distingue plusieurs subtilités. Les opérateurs * et -> ont été redéfinis pour une utilisation transparente du mécanisme des smart pointeurs (pointeurs intelligents).

Membres static et les templates

Il est possible de mettre des membres static dans les templates de classes.

```
template <typename T> class Foo {
public:
  static std::size_t count() { return ctr; }
private:
  static std::size_t ctr;
};
```

Tous les objets de type Foo<T> partagent les mêmes données static.

```
// instantiation des membres static Foo<string>::ctr and Foo<string>::count
Foo<string> fs;
```

```
// tous les 3 objets partagent les membres Foo<int>::ctr and Foo<int>::count
Foo<int> fi, fi2, fi3;
```

Templates et arguments par défaut

Il est possible de passer des types par défaut pour les paramètres de type d'un template.

```
template <typename T, typename F = less<T>>
int compare(const T &v1, const T &v2, F f = F())
{
  if (f(v1, v2)) return -1;
  if (f(v2, v1)) return 1;
  return 0;
}
```

Ici le deuxième paramètre du template est par défaut et c'est less<T>. La fonction less de la STL prend deux paramètres (x et y) et retourne x<y.

Spécialisation de template

Dans certains cas précis, on est obligé de demander un fonctionnement particulier du template pour un type de paramètre précis. Prenons un exemple :

```
template <typename T>
struct Wrap
{
    typename typedef T type;
    static const T& MakeWrap(const T& t)
    {
        return t;
    }
    static const T& Unwrap(const T& t)
    {
        return t;
    }
};
```

Ce template Wrap<T> représente un wrapper universel. Par défaut MakeWrap et Unwrap retourne T et le typedef type aussi retourne un T. Pour l'ensemble des types simples, ce wrapper fonctionne mais pour les type HSTRING et les pointers, il faut wrapper cela autrement. Ce développement est spécifique Windows 8.

```
template <>
struct Wrap<HSTRING>
{
    typedef HStringHelper type;
    static HStringHelper MakeWrap(const HSTRING& t)
    {
        HStringHelper str;
        str.Set(t);
        return str;
    }
    static HSTRING Unwrap(const HStringHelper& t)
    {
        HSTRING str;
        t.CopyTo(&str);
        return str;
    }
};

template <typename T>
struct Wrap<T*>
{
    typename typedef ComPtr<T> type;
    static ComPtr<T> MakeWrap(T* t)
    {
        ComPtr<T> ptr;
        ptr.Attach(t);
        return ptr;
    }
    static T* Unwrap(ComPtr<T> t)
    {
        ComPtr<T> ptr(t);
        return t.Detach();
    }
};
```

Le type HSTRING ne peut pas être manipulé tel-que car c'est comme un HANDLE. C'est la raison pour laquelle HSTRING est traité avec une classe HStringHelper et type devient HStringHelper Il en va de même pour les pointeurs qui doivent être encapsulés dans des ComPtr<T> car ce sont des composants COM. Le type est ComPtr<T> car les pointeurs sont des interfaces COM et il faut les encapsuler pour les utiliser. Oui c'est du développement Windows mais l'essentiel c'est de remarquer la spécialisation du template Wrap<T> et ses variantes Wrap<HSTRING> et Wrap<T*>.
Le concept clé c'est que les règles standard s'appliquent aux spécialisations. Pour spécialiser un template, une déclaration du template original doit être accessible. Il en va de même pour le template spécialisé avant son utilisation.
Avec des classes ou des fonctions ordinaires, les déclarations manquantes sont faciles à trouver. Pour les templates, c'est souvent plus compliquer. Surtout si la spécialisation du template n'est pas contenue dans le même fichier d'entête que le template principal. Les templates et les templates spécialisés doivent être déclarés dans le même fichier d'entête. En premier lieu, on définit le template original et ensuite les templates spécialisés.

Facilités de déclaration dans les templates

Dans certains cas, le typedef sur un type va permettre une meilleure lisibilité du code. Exemple :

```
template <class T>
class Iterator : public RuntimeClass<IIterator<T>>
{
    InspectableClass(L"Library1.Iterator", BaseTrust)

private:
    typedef typename std::vector<typename Wrap<T>::type> WrappedVector;
    typedef typename std::shared_ptr<WrappedVector> V;
    typedef typename WrappedVector::iterator IT;

 ../..
private:
    V _v;
    IT _it;
    T _element;
    boolean _bElement = FALSE;
};
```

Ce template Iterator<T> contient un WrappedVector qui est un typedef sur vector<typename Wrap<T> ::type>.
Le fait d'utiliser Wrap<T>::type fait la magie de ce template. Pour des types simples, T vaut T et pour HSTRING et T*, ça vaudra HStringHelper et ComPtr<T>. Ce mécanisme est très puissant.
Dans cet exemple, le template Iterator<T> représente un itérateur sous Windows 8 en mode Windows Store. Oui je sais c'est du Windows mais c'est le style qui compte. Regardez les typedef, ils utilisent Wrap<T> dans un vector<T> et l'itérateur du vector est défini en IT, plus simple à écrire.
La déclaration de V est un shared_ptr de vector<T> avec T qui vaut Wrap<T>. C'est assez complexe mais cela veut dire que c'est un container générique d'objets wrappés qui peut contenir à la fois des types simples mais aussi des HSTRING et aussi des pointeurs d'interfaces T* qui sera géré comme des pointeurs d'interfaces COM donc encapsulé avec la spécialisation de Wrap<T*> via ComPtr<T>.
Regardons comment est définie la méthode GetCurrent pour cette classe itérateur.

```
        virtual HRESULT STDMETHODCALLTYPE get_Current(T *current)
        {
            try {
                _LogInfo(L"Iterator::get_Current()...");
            if (_it != _v->end())
            {
                _bElement = TRUE;
                _element = Wrap<T>::Unwrap(*_it);
                *current = _element;
            }
            else
            {
                _bElement = FALSE;
            }
            return S_OK;
            _EXCEPTION_HANDLER(L"Iterator::get_Current()...");
        }
```

GetCurrent doit retourner l'élément courant ; on vérifie l'itérateur pour savoir si on est à la fin ou pas. On Unwrap l'élément et on le retourne dans le paramètre de la fonction. Le booléen _bElement est positionné pour avoir une information en double sur le statut de position de l'itérateur. Il est utile pour une autre méthode du template pour savoir s'il existe un élément courant.
Vous allez me dire, OK on déclare un itérateur mais où est la classe du container qui utilise cet itérateur. Voici la classe Vector<T> spécifique pour Windows 8 qui permet de gérer un container générique en prenant soin des types simples et des types complexes (HSTRING, pointeurs d'interface COM) avec Wrap<T>.

```
template <typename T>
    class Vector : public RuntimeClass<IVector<T>,
        IIterable<T>,
        IObservableVector<T>>
    {
        InspectableClass(L"Library1.Vector", BaseTrust)

    private:
        typedef typename std::vector<typename Wrap<T>::type> WrappedVector;
        typedef typename WrappedVector::const_iterator CIT;
        typedef typename VectorChangedEventHandler<T> WFC_Handler;

    public:
        Vector()
        {
            _LogInfo(L"Vector::Vector()...");
            _v = std::make_shared<WrappedVector>();
            m_observed = false;
        }
    public:
        virtual HRESULT STDMETHODCALLTYPE GetAt(unsigned index, T *item)
        {
            _LogInfo(L"Vector::GetAt()...");
            *item = Wrap<T>::Unwrap((*_v)[index]);
            return S_OK;
        }

        ../..
        virtual HRESULT STDMETHODCALLTYPE First(IIterator<T> **first)
        {
            _LogInfo(L"Vector::First()...");
            ComPtr<IIterator<T>> p = Make<IIterator<T>>();
            p->Init(_v);
            *first = p.Detach();
            return S_OK;
        }
        ../..
private:
        std::shared_ptr<WrappedVector> _v;
        bool m_observed;
        EventSource<VectorChangedEventHandler<T>> m_events;
    };
```

Pour simplifier la compréhension de ce template Vector<T>, je n'ai fait figurer que la méthode GetAt qui permet de récupérer un élément et la méthode First qui retourne un itérateur sur le Vector<T> via Iterator<T>, template que nous avons découvert précédemment. Le template Vector<T> hérite de plusieurs classes qui sont aussi des templates.

Conclusion

Les templates sont des classes ou des fonctions qui permettent de générer des classes. Leur utilisation nécessite un peu de pratique mais c'est quelque chose d'appréhensible avec un peu d'effort. La puissance des templates réside dans la possibilité de définir une classe générique et de prévoir les adaptations nécessaires pour qu'elle puisse être utilisée avec le minimum de contraintes. La spécialisation partielle des templates est une formidable mécanique pour corriger les types un peu limités ou trop pénibles à gérer. Dans ces opérations, le typedef est ton ami. On déclare un T et puis avec son utilisation on se rend compte que T est trop simple et qu'il faut wrapper le T dans certains cas. C'est cool. Cet article vous permet de découvrir ce que sont les templates. Cela peut paraître compliqué mais avec un peu de pratique, on est complètement aspiré et tout devient limpide. Il faut toujours se mettre en situation avec la casquette du designer du template et changer de temps en temps en tant qu'utilisateur du template. La découverte des erreurs de compilations les plus complexes se fera via le code d'utilisation des templates. Enjoy ! Le code source Windows 8 autour de Vector<T>, Iterator<T>, IIterable<T> et Wrap<T> est accessible ici : http://code.msdn.microsoft.com/windowsapps/Windows-Runtime-Component-4dc6fa20