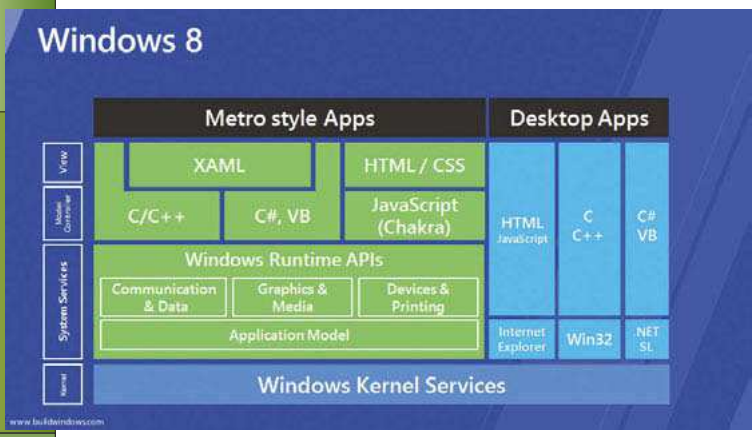
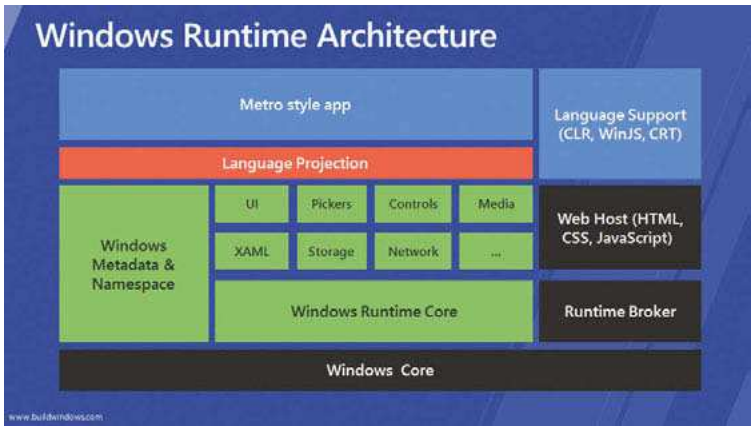
Ecrire des composants WinRT en C++ avec WRL

Depuis la sortie de Windows 8, Microsoft a proposé un ensemble de nouveaux outils et de technologies pour le développement des applications « Windows Metro Style Apps ». Dans le monde Metro qui est l'ancien nom marketing de ces applications disponibles sur le Microsoft Store, on parle d'interfaces XAML, du Windows Runtime et de composants WinRT. Si vous êtes habitué au développement Windows traditionnel, vous connaissez les API Windows Win32. Dans le monde des « Windows Store Apps » qui est la nouvelle appellation de ces applications pour le Windows Store, tout se concentre autour d'une brique technologique qui est le Windows Runtime. Cette brique et son architecture ont été présentées à la conférence BUILD 2011 de Microsoft suite à la sortie beta de Windows 8.





Le Windows Runtime se compose de composants techniques et d'un ensemble d'API que sont les Windows Runtime APIs. La programmation des applications Windows Store Apps est faite de l'utilisation de ces APIs. Ces APIs sont disponibles sous formes de composants WinRT qui s'exécutent sur un socle Windows standard.

Les technologies des composants COM et OLE2

Historiquement, les composants COM sont une technologie conçue par Microsoft pour l'échange de composants binaires entre différents langages comme Visual C++ et Visual Basic. La technologie COM est à la base de la technologie OLE2 qui a aussi été nommée par un terme marketing plus général qu'est ActiveX. Les modèles COM et OLE2 fournissent plusieurs interfaces pour créer des composants plus ou moins complexes, exposer et échanger des propriétés, des méthodes, des évènements, des contrôles graphiques réutilisables et des mécanismes de communication. Les deux interfaces les plus connues de ce modèle de composants sont IUnknown et IDispatch. Les composants COM doivent être enregistrés dans la base de registre locale du poste.

Un composant WinRT c'est quoi ?

Pour Windows 8, Microsoft étend la technologie COM. En effet, un composant WinRT est un composant COM de nouvelle génération. Une nouvelle interface fait son apparition. Elle est disponible dans le fichier inspectable.idl. Cette interface hérite de IUnknown et fournit 3 nouvelles méthodes que sont Getlids, GetRuntimeClassName et GetTrustLevel :

```
interface IInspectable : IUnknown
{
  HRESULT GetIids(
    [out] ULONG * iidCount,
    [out, size_is(,*iidCount)] IID ** iids);

  HRESULT GetRuntimeClassName( [out] HSTRING * className);

  HRESULT GetTrustLevel([out] TrustLevel * trustLevel);
}
```

Dans le modèle COM traditionnel, les interfaces et composants sont décrits dans des fichiers IDL et sont compilés par le compilateur MIDL pour produire des fichiers d'entête .h et .c pour le monde C/C++ et des fichiers TLB (Type Library) pour les autres langages. Avec les composants WinRT, la logique est conservée.

On décrit les interfaces et les composants (appelés runtimeclass au lieu de coclass) dans un fichier IDL et on compile avec MIDL.

Le compilateur MIDL produit des fichiers .h et .c mais aussi des Windows metadata (.winmd) comparables à ceux qui existent dans le monde du .NET Framework. Il est possible d'ouvrir les fichiers d'extension winmd dans l'outil ILDASM pour explorer les interfaces, les classes, les méthodes et les propriétés.

Donc un composant WinRT, c'est un composant qui implémente IInspectable et qui possède des Windows metadata. C'est tout !

Ecrire un composant WinRT en C++

Il existe deux possibilités en C++ pour développer des composants WinRT :

- Utiliser le C++/CX
- Utiliser la librairie WRL (Windows Runtime Library) disponible dans le Windows SDK

Introduction à C++/CX

Le C++/CX est un ensemble d'extensions du compilateur Visual C++ pour écrire des composants WinRT. C++/CX possède une syntaxe et des mots clés que ne possède pas le C++ traditionnel avec des ^et des % comme le C++/CLI. Cependant le code généré est natif ; ce n'est pas du .Net comme le C++/CLI. Un composant simple en C++/CX ressemble à cela :

```
namespace MyWinRTComponent
{
 public ref class Logger sealed
 {
 public:
   Logger() { }
 public:
   void LogInfo(String^ message) { … }
 };
}
```

Il suffit de déclarer une classe avec les mots clés publics ref et sealed et toutes méthodes publiques seront exposées. Le but de cet article n'est pas d'illustrer C++/CX car ce dérivé de C++ n'est pas le vrai C++ ISO. Il est peut-être intéressant pour des nouveaux développeurs, mais son adoption n'est pas un franc succès et la communauté C++ est attachée au vrai C++, celui qui est défini comme un standard ISO. Il existe une autre possibilité qui est franchement plus intéressante. Microsoft l'utilise pour faire les composants qui sont livrés avec Windows 8. Cette autre façon consiste à utiliser la librairie WRL qui a pour but de couvrir tous les principes internes exposés par le Windows Runtime. Le Windows Runtime est fabriqué avec la librairie WRL alias *Windows Runtime Library*.

Utiliser la librairie Windows Runtime Library (WRL)

WRL est disponible dans le Windows SDK de Windows 8 et Windows 8.1 sous C:\Program Files (x86)\Windows Kits\8.x\Include\winrt\wrl.

Fichier	Usages principaux
async.h	Gestion des mécanismes asynchrones
client.h	Classes ComPtr, WeakRef
def.h	Internes
event.h	fonction Callback()
ftm.h	internes
implements.h	RuntimeClass, ModuleBase
internal.h	internes
module.h	Module
wrappers\corewrappers.h	HString, HStringReference

Cette librairie est construite sur le même modèle que son ancêtre ATL. ATL est disponible dans Visual C++ pour faire des composants COM. WRL est disponible pour faire des composants WinRT. Pour faire un simple composant, il faut d'abord créer un fichier IDL qui va décrire les interfaces et les composants. Voici un composant simple dans un fichier IDL :

```
// Library1.IDL
import «inspectable.idl»;
import «Windows.Foundation.idl»;

#define COMPONENT_VERSION 1.0

namespace Library1
{
 interface ILogger;
 runtimeclass Logger;

 [uuid(3EC4B4D6-14A6-4D0D-BB96-31DA25224A15), version(COMPONENT
_VERSION)]
 interface ILogger : IInspectable
 {
   [propget] HRESULT Name([out, retval] HSTRING* value);
   [propput] HRESULT Name([in] HSTRING value);
   HRESULT LogInfo([in] HSTRING value);
 }

 [version(COMPONENT_VERSION), activatable(COMPONENT_VERSION)]
 runtimeclass Logger
 {
   [default] interface ILogger;
 }

}
```

Chaque interface et chaque composant ou runtimeclass requiert un identifiant unique qui est un GUID. Un composant WinRT implémente une ou plusieurs interfaces. Ces interfaces exposent des méthodes et/ou des propriétés, mais ne peuvent exposer que des types Windows. Pour échanger des types compatibles avec les langages comme C# et VB.NET, il faut utiliser des interfaces fournies par le Windows Runtime pour qu'il puisse mettre en œuvre les projections pour chaque langage. Pour échanger des types simples comme int, long, float ou double, il n'y a pas de problèmes.

Pour le type chaîne de caractères, il faut utiliser le type HSTRING. Ce type sera mappé automatiquement sur le type String en C#. Ce type HSTRING remplace le type BSTR qui existait autrefois pour exposer des chaînes de caractères dans le monde Visual Basic et OLE2. Il existe une classe qui encapsule ce type Windows et qui se nomme HString. Elle permet de travailler avec des types wchar_t.

Un composant WinRT avec WRL

Maintenant que l'interface et le composant sont définis dans le fichier IDL, il faut fournir une implémentation du composant. Nous allons tirer parti des classes et templates de WRL :

```
#include «Library1_h.h»

namespace ABI
{
 namespace Library1
 {
   class Logger : public RuntimeClass<ILogger>
   {
     InspectableClass(L»Library1.Logger», BaseTrust)

   public:
     STDMETHOD(get_Name)(HSTRING* value) { return S_OK; }
     STDMETHOD(put_Name)(HSTRING value) { return S_OK; }
     STDMETHOD(LogInfo)(HSTRING value) { return S_OK; }
   };

   ActivatableClass(Logger);
 }
}
```

La classe se trouve dans le namespace ABI. C'est une convention Microsoft. La classe hérite de `RuntimeClass<ILogger>` et le template RuntimeClass prend en argument l'interface que nous avons définie. La classe du composant utilise deux macros :

- InspectableClass qui contient le nom du composant et fournit l'implémentation de IInspectable,
- ActivatableClass qui fournit la factory nécessaire à cette classe.

Comme tous les composants COM, les différentes méthodes retournent un HRESULT. Contrairement aux composants COM traditionnels, les composants WinRT n'ont pas besoin d'être enregistrés dans la base de registres. Ils doivent juste être accessibles au travers d'une class-factory. C'est la macro `ActivatableClass` qui fournit ce service via la classe `SimpleSealedActivationFactory`.

Il faut cependant un peu de code pour la plomberie WinRT. Si vous téléchargez le project template WRL, ce code sera ajouté automatiquement à votre projet WRL. Voici le fichier module.cpp qui ressemble à cela après quelques modifications mineures :

```
// module.cpp : Defines the module that contains the com classes
//
#include «pch.h»
#include <wrl\module.h>

extern «C» HRESULT WINAPI DllGetActivationFactory(_In_ HSTRING
 activatibleClassId, _Deref_out_ IActivationFactory** factory)
{
 auto &module = Microsoft::WRL::Module<Microsoft::WRL::InProc>
::GetModule();
 return module.GetActivationFactory(activatibleClassId, factory);
}

extern «C» HRESULT WINAPI DllGetClassObject(REFCLSID rclsid,
 REFIID riid, _Deref_out_ LPVOID *ppv)
{
 auto &module = Microsoft::WRL::Module<Microsoft::WRL::InProc>
::GetModule();
 return module.GetClassObject(rclsid, riid, ppv);
}

extern «C» BOOL WINAPI DllMain(_In_opt_ HINSTANCE, DWORD, _In
_opt_ LPVOID)
{
 return TRUE;
}

extern «C» HRESULT WINAPI DllCanUnloadNow()
{
 const auto &module = Microsoft::WRL::Module<Microsoft::WRL
::InProc>::GetModule();
 return module.GetObjectCount() == 0 ? S_OK : S_FALSE;
}
```

Les routines `DllGetActivationFactory` et `DllGetClassObject` font partie de la plomberie nécessaire pour WinRT.

Gestion des collections comme les vecteurs

Pour avoir une méthode qui retourne une collection d'objets comme un vecteur ou une map, le sujet devient beaucoup plus compliqué. En effet, Microsoft fournit un support étendu pour C++/CX et fournit un adaptateur pour faire le lien entre les containers de la STL et les interfaces Windows Runtime au travers d'un fichier d'entête nommé collection.h et disponible depuis C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\include. Ce fichier se définit comme « `Windows Runtime Collection/Iterator Wrappers` ». C'est un ensemble de wrappers. Il va nous fournir des facilités pour passer du monde Windows Runtime au monde STL avec les containers STL comme std::vector<T>, std::map<K,T> et std::unordered_map<K,T>. Cependant, il utilise une syntaxe particulière au C++/CX. Dans notre cas, en véritable C++ ISO, il nous faut de vraies classes ISO C++. Nous devons étudier les interfaces du Windows Runtime à implémenter et fournir nos propres wrappers. Si nous voulons exporter une collection d'objets à base de std::vector<T>, il nous faut implémenter les interfaces suivantes présentes dans le namespace `Windows::Foundation::Collections` que sont `IVector<T>`, `IIterable<T>` et `IIterator<T>`.

Ainsi, du côté du consommateur du composant en XAML C#, par exemple, la collection IVector<T> est exposée en IList<T> par projection.

```
// C# client
Library1.Root root = new Library1.Root();
IList<String> list = root.GetVector();
foreach(string s in list)
{ ... }
```

En étudiant ces 3 interfaces ci-dessus, on se rend compte qu'il faut définir chaque méthode d'accès au container. Examinons les internes de IVector<T> qui sont définies dans le répertoire C:\Program Files (x86)\Windows Kits\8.x\Include\winrt et dans le fichier windows.foundation.collections.h. Ce fichier contient les différentes interfaces. Voici avec quelques modifications ce que représente IVector<T> :

```
template <class T>
class IVector : IInspectable /* requires IIterable<T> */
{
public:
 // read methods
 virtual HRESULT GetAt(unsigned index, T *item) = 0;
 virtual HRESULT get_Size(unsigned *size) = 0;
 virtual HRESULT GetView(IVectorView<T> **view) = 0;
 virtual HRESULT IndexOf(T value, unsigned *index, boolean
*found) = 0;
 // write methods
 virtual HRESULT SetAt(unsigned index, T item) = 0;
 virtual HRESULT InsertAt(unsigned index, T item) = 0;
 virtual HRESULT RemoveAt(unsigned index) = 0;
 virtual HRESULT Append(T item) = 0;
 virtual HRESULT RemoveAtEnd() = 0;
 virtual HRESULT Clear() = 0;
 // bulk transfer methods
 virtual HRESULT GetMany(unsigned startIndex, unsigned capacity,
 T *value, unsigned *actual) = 0;
 virtual HRESULT ReplaceAll(unsigned count, T *value) = 0;
};
```

D'après la documentation, l'interface IVector<T> hérite de IIterable<T> et sa représentation ressemble à cela :

```
template <class T>
class IIterable : IInspectable
{
public:
```

```
  virtual HRESULT First(IIterator<T> **first) = 0;
};
```

Et l'interface IIterable<T> retourne au travers de la méthode First() une interface IIterator<T> dont la représentation ressemble à cela :

```
template <class T>
class IIterator : IInspectable
{
public:
  virtual HRESULT get_Current(T *current) = 0;
  virtual HRESULT get_HasCurrent(boolean *hasCurrent) = 0;
  virtual HRESULT MoveNext(boolean *hasCurrent) = 0;
  virtual HRESULT GetMany(unsigned capacity, T *value, unsigned
  *actual) = 0;
};
```

Cette notion d'itérateur est semblable à l'itérateur du monde C++ de la STL. On possède un itérateur sur un container particulier. Le principe dans WinRT est le même. Maintenant que nous connaissons les interfaces et leurs méthodes, il faut les implémenter.

Nous allons définir un template construit autour du container STL std::vector<T> et faire le mapping entre l'interface IVector<T> et les méthodes de std::vector<T>. On remarque que la correspondance est assez naturelle pour la plupart des méthodes et ressemble presque à cela :

- IVector<T>.GetAt -> std::vector<T>.operator [index]
- IVector<T>.get_Size -> std::vector<T>.size()
- IVector<T>.InsertAt -> std::vector<T>.emplace(begin() + index, item)
- IVector<T>.RemoveAt -> std::vector<T>.erase(begin() + index)
- IVector<T>.Append -> std::vector<T>.emplace_back(item)
- IVector<T>.RemoveAtEnd -> std::vector<T>.pop_back()
- IVector<T>.Clear -> std::vector<T>.clear()
- IIterator<T>.get_Current -> *iterator
- IIterator<T>.get_HasCurrent -> bool
- IIterator<T>.MoveNext -> ++iterator

J'ai créé 2 templates pour implémenter ces 3 interfaces. Ces templates sont intrinsèquement liés à std::vector<T>. Le code de ces 2 templates est trop important pour figurer dans l'article, mais vous pouvez le récupérer en lien à la fin de cet article et aussi sur le Dev Center Windows Store App dans un code nommé « Windows Runtime Component using WRL and C++ for Async and Vector Collection ».

Voici la structure de ces 2 templates :

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

    public:
      Vector()
      {
        std::shared_ptr<std::vector<T>> ptr(new std::vector<T>());
        _v = ptr;
      }

…

    private:
      std::shared_ptr<std::vector<T>> _v;
    };
```

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

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

      …

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

Le template Vector<T> implémente les interfaces IVector<T> et IIterable<T> tandis que Iterator<T> implémente l'interface IIterator<T>. J'ai volontairement encapsulé le std::vector<T> dans un shared_ptr<T> pour les besoins de WinRT. En effet, il est possible de fournir une vue sur le vecteur et aussi un mode observateur sur le vecteur. C'est indispensable à implémenter si vous voulez supporter le binding.

Mais ce sujet dépasse le cadre de cet article. L'essentiel est d'avoir la mécanique qui permette de déclarer un vecteur de la STL et de l'exposer comme un type reconnu par WinRT. Maintenant que nous avons le wrapper pour retourner un IVector<T>, nous devons fournir un composant qui l'utilise et retourne la collection d'objets. Voici le composant Root qui possède deux méthodes : GetVector et GetVectorInt. Ces deux méthodes vont utiliser le même template pour retourner un IVector<T>. Voici la description IDL de Root et de son interface IRoot :

```
  [uuid(3EC4B4D6-14A6-4D0D-BB96-31DA25224A16), version(COMPONENT
_VERSION)]
  interface IRoot : IInspectable
  {
    HRESULT GetVector([out][retval] Windows.Foundation.Collections.
IVector<HSTRING>** value);
    HRESULT  GetVectorInt([out][retval]  Windows.Foundation.
Collections.IVector<int>** value);
  }

  [version(COMPONENT_VERSION), activatable(COMPONENT_VERSION)]
  runtimeclass Root
  {
    [default] interface IRoot;
  }
```

Voici la classe C++ du composant dans Root.h :

```
namespace ABI
{
  namespace Library1
  {
    class Root : public RuntimeClass<IRoot>
    {
      InspectableClass(L»Library1.Root», BaseTrust)

    public:
```

```
    Root() {}

  public:
    STDMETHOD(GetVector)(IVector<HSTRING>** value);
    STDMETHOD(GetVectorInt)(IVector<int>** value);
  };

  ActivatableClass(Root);
 }
}
```

Voici l'implémentation de la classe C++ du composant dans Root.cpp :

```
#include «pch.h»
#include «Root.h»
#include «Utility.h»
#include «Logger.h»
#include «Collection.h»

namespace ABI
{
 namespace Library1
 {
  STDMETHODIMP Root::GetVector(IVector<HSTRING>** value)
  {
   ComPtr<Vector<HSTRING>> v = Make<Vector<HSTRING>>();
   HString str;
   str.Set(L»String1»);
   v->Append(str.Detach());
   str.Set(L»String2»);
   v->Append(str.Detach());
   str.Set(L»String3»);
   v->Append(str.Detach());
   str.Set(L»String4»);
   v->Append(str.Detach());
   *value = v.Detach();
   return S_OK;
  }

  STDMETHODIMP Root::GetVectorInt(IVector<int>** value)
  {
   ComPtr<Vector<int>> v = Make<Vector<int>>();
   v->Append(10);
   v->Append(20);
   v->Append(30);
   v->Append(40);
   *value = v.Detach();
   return S_OK;
  }
 }
}
```

La fonction Make<> permet de créer un objet ComPtr<T>. On retourne cet objet avec la méthode Detach(). C'est la même mécanique qu'au temps de la librairie ATL. Il est possible de retourner d'autres types de collections que les vecteurs.

On peut retourner une map représentée par l'interface IMap<K,V> qui est exposée en projection au monde C# par IDictionary<K,V>. Cependant, il faut fournir le template qui « wrap » le container std::map<K,V>. Eh oui, il reste des choses à construire...

Les méthodes asynchrones

C'est un pattern largement utilisé dans les Windows Runtime APIs. Les méthodes async s'exécutent sans bloquer le thread d'appel. Dans le monde C#, l'utilisation d'une méthode async se fait avec l'utilisation du mot clé await. Nous allons créer un composant Root qui retourne un objet Logger au travers son interface ILogger. Voici le code client XAML C# client qui fait appel à notre composant WinRT :

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
 Library1.Root root = new Root();
 Library1.ILogger logger = await root.GetLoggerAsync();
 logger.LogInfo(«LogInfo()...»);
}
```

Comme pour la gestion des collections, la gestion des méthodes async est facile à réaliser avec C++/CX. En effet, la librairie PPL Taks a été modifiée pour supporter la création de tâche avec le pattern async. Ce pattern met en œuvre plusieurs interfaces du Windows Runtime. Pour le monde du vrai C++ ISO, la librairie WRL fournit la classe AsyncBase<T> qui fournit une implémentation de la machine à état nécessaire pour le fonctionnement des méthodes async. La mauvaise nouvelle est que AsyncBase n'est pas documentée... Il suffit de faire une recherche sur les mots « WRL AsyncBase ». Avec Bing, on obtient 14 résultats ; un record ! Pour tirer parti de cette classe, il suffit de réaliser une classe dérivée de la classe AsyncBase et de fournir une surcharge de certaines méthodes virtuelles. Pour fournir la mécanique d'exécution asynchrone, nous allons utiliser PPL (Parallel Patterns Library) et la class task<T>. Pour comprendre la mécanique async du Windows Runtime, il faut s'intéresser aux interfaces suivantes :

- Windows::Foundation::IAsyncOperation<TResult>
- Windows::Foundation::IAsyncOperationWithProgress<TResult, TProgress>
- Windows::Foundation::IAsyncAction
- Windows::Foundation::IAsyncActionWithProgress<TProgress>

Ces quatre interfaces héritent de Windows::Foundation::IAsyncInfo. Pour créer une méthode async, je vais utiliser un template qui tire parti de AsyncBase et qui exécute la fonction passée en paramètre dans une task de PPL. Voici à quoi ressemble le composant Root qui retourne un Logger de manière async :

```
STDMETHODIMP Root::GetLoggerAsync(IAsyncOperation<ILogger*>
** value)
{
 ComPtr<task_based_async_operation<ILogger>> pObject =
Make<task_based_async_operation<ILogger>>(
   std::async([&]() -> ILogger*
 {
   ComPtr<Logger> p = Make<Logger>();
   return p.Detach();
 }));
 *value = pObject.Detach();
 return S_OK;
}
```

Le template classe task_based_async_operation prend en paramètre l'interface de retour (ici ILogger) de la fonction async. Dans notre exemple, la task PPL qui est créée ne fait que construire un objet Logger sachant que cet objet implémente l'interface ILogger.

Code source des templates C++ présents dans cet article

Vous pouvez télécharger le code source des templates C++ sur le « Dev Center - Windows Store App » en faisant une recherche sur « Windows Runtime Component using WRL and C++ for Async and Vector Collection » ou bien taper l'URL suivante : http://code.msdn.microsoft.com/windowsapps/Windows-Runtime-Component-4dc6fa20

Ce code vous permet de démarrer avec un projet de dll Win32 pour WinRT.

Demandez plus aux équipes Microsoft US

J'ai eu quelques échanges avec l'équipe Visual C++ de Microsoft et je leur ai fait part de ma surprise de constater que seul le support de C++/CX est réellement bien documenté par le biais d'exemples en standard dans le produit Visual C++, mais, que concernant C++ et WRL, il n'existe toutefois qu'une description sommaire des classes. En effet, les développeurs de Microsoft utilisent la librairie WRL pour faire Windows 8 et ses composants WinRT; WRL peut être considéré comme le successeur de ATL. J'ai expliqué que j'avais eu du mal à trouver des exemples ou de la documentation sur IVector<T> et `windows.foundation.collections.h`. Pour la gestion des méthodes async par exemple, c'était la solitude complète... Bing m'a retourné 14 résultats sur AsyncBase. Pour la gestion des collections à base de IVector<T>, j'ai bien vu des choses dans le code de Mozilla Firefox pour Windows RT et celui de Google Chromium pour WinRT qui m'ont confirmé que ce n'est que de l'implémentation standard d'interfaces COM ; c'est navrant de devoir aller dans leur répository Git ou Svn pour trouver des indices sur les technologies Microsoft... Pour le moment, les échanges avec les US sont limités mais Microsoft est à l'écoute de la communauté C++... On m'indique que le code généré pour C++/CX est optimisé pour la performance. Oui, peut-être, mais écrire une classe en C++ avec WRL n'est pas très dur, et je peux utiliser du C++ ISO, ce qui est très rassurant plutôt que les extensions du compilateur Visual C++ et sa syntaxe ^ et % qui n'a rien à voir avec celle du C++ standard. Microsoft utilise WRL en interne et ça me rassure. Avec du code C++ standard, je peux réutiliser mon code existant, je peux le porter sur d'autres plateformes facilement en « wrappant » les templates et macros avec des coquilles vides. Bref, je ne me ferme aucune porte. Et puis j'avoue que j'ai un faible pour la technique donc je préfère maitriser mon code plutôt que d'être dépendant d'un compilateur. Le C++ donne le pouvoir et la performance. Si comme moi, vous trouvez que lire les entêtes du Windows Kits pour comprendre les templates C++ et les interfaces du Windows Runtime devrait être accompagné d'un peu plus de documentation et d'exemples de code, manifestez-vous et ils répondent. Voici les contacts à utiliser :

- Visual C++ General Manager : Alessandro.Contenti@microsoft.com
- Visual C++ Lead : Tarek.Madkour@microsoft.com
- L'auteur de WRL : Sridhar.S.Madhugiri@microsoft.com

Je dois vous avouer qu'il existe d'autres éléments à creuser comme par exemple la notion du binding, les collections observables, les events et les delegate. Bref, si vous êtes un programmeur C++ curieux, vous avez la possibilité de comprendre les internes du Windows Runtime rien qu'en regardant et en lisant les fichiers d'entêtes du Windows Kits qui sont des templates C++. Avouez que c'est bien plus fun que de faire du C# sans rien comprendre des couches sur lesquelles on s'appuie ? A méditer...

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