C++ 11 : une version aboutie

Le C++ est un langage vivant. Pour le prouver, si le standard ISO C++11 est disponible depuis trois ans (mars 2011), la roadmap pour les futurs standards C++14 et C++17 a été annoncée récemment par Herb Sutter, le secrétaire de standardisation du langage C++ de l'ISO. Le comité ISO a en effet décidé de délivrer de nouvelles versions du standard plus rapidement, que ce soit au niveau des extensions du langage, que de la librairie STL (Standard Template Library).

Le support de C++11 dans les compilateurs

Le support du C++11 par les principaux compilateurs du marché - GCC, CLang et Visual C++ - est maintenant assuré en totalité. L'implémentation des fonctionnalités ne s'est pas faite en une seule fois; vérifiez bien le support de conformité de votre compilateur si vous n'utilisez pas la dernière version. C++11 regroupe à la fois le langage C++ et la librairie STL. La librairie STL exploite certaines nouvelles fonctionnalités du langage (la sémantique de déplacement par exemple).

C++11 et STL

En C++ moderne, l'utilisation de la librairie STL est implicite car elle fournit, par exemple, les classes string et les « containers » comme vector, list, set, map et le support de tables de hash. Il est admis que ces collections sont ce qui se fait de fait de plus efficace. La STL fournit aussi tout un ensemble d'algorithmes pour le parcours ou la transformation des ensembles de données.

Les lambdas en C++

Le C++ dispose maintenant de la fonctionnalité des lambdas. Une lambda est une fonction anonyme qui n'est pas nommée qui peut prendre des arguments et qui possède un corps de fonction et un type de retour. Le type de retour par défaut est void, mais si vous devez définir un autre type de retour, il faut préciser ce type avec la syntaxe de la flèche ->. En plus des arguments passés à la fonction, une lambda définit une liste de capture décrite entre []. Voici une fonction lambda simple utilisée avec une opération for each :

```
void Use_Lambda()
{
  vector<string> v;
  v.push_back(«Edith»);
  v.push_back(«Lisa»);
  v.push_back(«Audrey Maggy»);

  std::for_each(v.begin(), v.end(), [](string str) -> void {
    cout << str << endl;
  });
}
```

Dans cet exemple, la liste de capture est vide. Si la syntaxe [&] est utilisée, toutes les variables locales seront passées en référence. Techniquement, une fonction lambda est définie dans une classe avec une surcharge de l'opérateur ().

Références RValue et move constructor

Le C++11 introduit la notion de référence rvalue, identifiée par T&&. Cette notion permet de mettre en œuvre la sémantique de déplacement (move semantic). Le résultat c'est que nous pouvons librement déplacer les ressources d'une référence rvalue vers un autre objet. Cette mécanique de déplacement est particulièrement utile pour éviter la copie de variable passée par valeur qui peut être couteuse en termes de performance. Le meilleur exemple consiste à retourner un vecteur créé dans une fonction. En C++03, les éléments du vecteur sont copiés les uns derrière les autres. En C++11, le vecteur est déplacé. Cette opération est supportée parce que le vecteur possède un constructeur de déplacement (move constructor) qui prend en paramètre une référence rvalue à un vecteur.

```
vector<string> Use_VectorReturn()
{
  vector<string> local_vector;
  local_vector.push_back(«Didou»);
  local_vector.push_back(«Liz»);
  local_vector.push_back(«Maggy»);

  printf(«local_vector=0x%08x\n», local_vector);
  return local_vector;
}

int _tmain(int argc, _TCHAR* argv[])
{
  vector<string> v = Use_VectorReturn();
  printf(«v=0x%08x\n», v);
  return 0;
}
```

La sortie de ce programme prouve que l'adresse du vecteur a été « volée ». Dans main, le vecteur pointe sur la même adresse que la variable locale de la fonction Use_VectorReturn().

```
local_vector=0x00590a00
v=0x00590a00
```

Dans notre exemple, le vecteur ne contient que très peu d'éléments. Dans la vraie vie, un vecteur peut contenir une quantité importante d'éléments et cette mécanique qui évite les copies locales permet de gagner en performance. Il faut noter que cette sémantique de déplacement est transparente pour l'utilisateur car les containers STL du C++11 disposent tous d'un constructeur de déplacement, mais aussi d'un opérateur de copie de déplacement (move-assignement operator). Une opération de déplacement exécute un « vol » de ressources et n'alloue aucune ressource. Ces opérations de déplacement n'ont pas vocation à lever des exceptions. Pour cela, nous devons le préciser en utilisant le nouveau mot-clé noexcept à la fois dans le fichier h et le fichier cpp. La librairie STL du C++11 introduit la fonction std::move() définie dans le fichier d'entête utility, qui permet de retourner un référence rvalue.

Expression constante

Le nouveau mot-clé constexpr garantit qu'une fonction ou que la construction d'un objet soit une opération constante garantie à la compilation. Le corps de la fonction doit contenir exactement une instruction return. Le compilateur remplace l'appel à une fonction constexpr directement par sa valeur.

Template marqué extern

Dans les applications importantes, l'instanciation d'un même template dans plusieurs fichiers peut rendre le temps de compilation important. Pour améliorer cela, il est possible de faire référence à un template dans différents fichiers sans pour autant que le compilateur ne génère le code nécessaire à l'instanciation de celui-ci. Il suffit alors de préfixer la définition du template comme extern.

Initialisation avec {}

Il est possible d'initialiser une variable ou un container STL comme vector, list ou map avec la syntaxe {} comme pour les structures en C :

```
void Use_Initialisation()
{
  int i = {0};
  vector<int> v = { 10, 20, 30, 40, 50 };
  list<string> mes_chats = { «Tom», «Pti'Gris», «Romeo» };
}
```

L'initialisation de liste est implémentée via l'utilisation de la classe std::initializer_list comme constructeur.

Utilisation de auto

Le mot clé auto est certainement la fonctionnalité qui va avoir le plus de succès en termes d'adoption.
L'utilisation de auto permet de laisser le compilateur déduire le type de la variable déclarée.

```
void Use_auto()
{
  vector<int> v = { 10, 20, 30, 40, 50 };
  //for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
  for (auto it = v.begin(); it != v.end(); it++)
  {
    int v = *it;
    cout << «value = « << v << endl;
  }
}
```

Le mot clé auto peut être utilisé avec const et &.

Evolution du for

Le parcours d'un ensemble de données avec for est disponible depuis que le C existe. C++11 introduit une variante pour le parcours des ensembles au travers de l'utilisation du mot clé for. La nouvelle syntaxe ressemble à cela :

```
void Use_RangeFor()
{
  vector<string> langs = { «VB6», «C#», «C++» };
  for (string s : langs )
  {
    cout << s << endl;
  }
}
```

Evolution du constructeur

C++11 introduit la notion de délégation de constructeur. En C++03, un constructeur non trivial affecte des valeurs à des variables membres. S'il existe plusieurs constructeurs, on se retrouve à dupliquer le code qui affecte les valeurs par défaut à certaines variables membres.

Classe marquée final

Si vous définissez une classe et que vous ne voulez pas qu'elle soit dérivée, il suffit d'ajouter le mot clé final après votre nom de classe. Si une classe essaie de dériver de celle-ci, le compilateur sortira en erreur.

```
class Bar final
{
};

// Illegal construction !!! Compilation error !!!
class DerivedBar : public Bar
{
};
```

Constante de pointeur nul

Par convention et par compatibilité avec le langage C, un pointeur nul vaut 0. La variable de préprocesseur NULL est utilisée et vaut 0. Le C++11 introduit le mot clé nullptr pour remplacer l'usage du NULL mais ne vaut pas 0 !

Template variadic

Un template variadic est une fonction ou classe template qui prend un nombre d'arguments variables. La syntaxe ... est utilisée pour définir les paramètres variables.

Séquences de chaines de caractères

La gestion de chaînes de caractères complexes pose un problème en C ou en C++. En effet, les séquences avec le caractère '\' posent problème. Le caractère '\' doit être doublé pour ne pas rentrer en conflit avec les séquences comme '\n', '\r', ou \'t'. De plus, on ne peut pas déclarer une chaîne de caractères sur plusieurs lignes. C++11 répond à ce problème en fournissant un support pour les « raw string literals ». Il est maintenant possible de déclarer le contenu d'un document XML directement dans le code. C++11 répond à cela avec une séquence qui ressemble à R»(« contenu »)»:

```
  string strXML1 = R»(<DataApollo>
  <Info>
    <PriceUnit>euro</PriceUnit>
    <VolumeUnit>TW0</VolumeUnit>
    <RunCode>2012-03_00_avantEBO_V1_20130110</RunCode>
    <RunOption>am</RunOption>
    <GenerationDate>24-02-2014</GenerationDate>
    <CoreVersion>2.3.3.894</CoreVersion>
    <ReturnCode>0</ReturnCode>
  </Info>
</DataApollo>
)»;
```

Les tuples

La librairie STL introduit la notion de tuple au travers la class std ::tuple. Cette implémentation est réalisée en utilisant les templates variadic. La classe std::tuple prend un nombre d'argument variables.

```
void Use_Tuple()
{
  tuple<string, string, int> maggy( «Maggy», «super coquine», 3);
  string str = get<1>(maggy);
  cout << str << endl;
}
```

Le type array

La classe array est définie dans le fichier d'entête array. Ce type permet de gérer une séquence d'éléments de taille fixe. C'est comme un tableau standard mais avec le support de la notion d'itérateur. Ce type est particulièrement adapté pour la programmation embarquée. Il supporte l'initialisation sous forme de liste :

```
void Use_Array()
{
  array<int, 10> ar = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
  for (auto it = ar.begin(); it != ar.end(); it++) {…}
  int a2 = ar[2]; int a3 = ar.at(3);}
```

Les hash tables

C++11 regroupe une seule et même notion de hash tables au travers des fichiers d'en-tête « unordered_set » et « unordered_map ». Les anciens fichiers « has_set » et « hash_map » sont obsolètes.

Les expressions régulières

Le support des expressions régulières est assuré au travers du fichier d'en-tête « regex ». Regex peut utiliser les grammaires des expressions régulières suivantes : ECMAScript, POSIX standard, étendu, awk, grep et egrep. Les fonctions regex_match() et regex_search() déterminent si une séquence de caractères correspond à une regex donnée. La fonction regex_replace() permet de trouver et remplacer une expression régulière dans une séquence. Regex utilise aussi la notion d'itérateur au travers la classe regex_iterator.

Les pointers intelligents (smart pointers)

Le support des « smart pointers » est assuré au travers du fichier d'en-tête « memory ». Les templates les plus utiles sont shared_ptr<T>, unique_ptr<T> et weak_ptr<T>. Le but avoué de cet effort de pointeurs intelligent est de supprimer l'usage du delete et de permettre la libération automatique de la mémoire. La gestion mémoire des objets peut être confiée aux pointeurs intelligents. Ils s'occuperont tout seul de la suppression des objets le temps venu. Le template shared_ptr<T> fonctionne avec l'ensemble des types C++ et ne nécessite aucune plomberie préalable dans le type qui souhaite en tirer parti. Le template shared_ptr<T> permet aussi, pour certains cas bien spécifiques, de spécifier la manière d'allouer ou de libérer le type T. Bref, il est possible d'allouer ou de libérer n'importe quel type de ressources. Pour gérer un pointeur intelligent, il suffit de d'encapsuler son type T avec le template shared_ptr<T> et d'utiliser la fonction make_shared<T>() à la place de new(). L'accès au type T se fait via l'opérateur -> donc le type T est utilisé comme un pointeur standard. Exemple :

```
void Use_SharedPtr()
{
  shared_ptr<MyClass> ptr = make_shared<MyClass>();
  ptr->data = «Maggy est une coquine !»;
  // no delete !!!
…
}
```

Dans cet exemple, la fonction delete n'est pas appelée, et le shared_ptr<T> s'en charge automatiquement. Il est possible d'affecter des shared_ptr et leur fonctionnement interne est basé sur le comptage de référence. Le compteur est automatiquement incrémenté ou décrémenté suivant les cas d'utilisation, et l'objet géré est libéré quand le compteur de référence tombe à 0.

Les threads et mutex

Le support des threads est assuré au travers du fichier d'entête thread. La classe thread encapsule le thread et sa routine de traitement associée en tirant parti du support des threads fournit par le système d'exploitation. Les API Linux et Windows sont différentes donc l'utilisation de std::thread permet d'uniformiser la programmation parallèle. La gestion des verrous est assurée au travers du fichier d'entête mutex. Un mutex est un objet qui représente un accès exclusif à une ressource. Il peut être utilisé pour protéger l'accès à une donnée et synchroniser l'accès aux données partagées entre plusieurs threads. Pour démarrer un thread, il suffit de lui passer le nom de la routine qui sera dédiée au traitement et éventuellement, une liste de paramètres à prendre en entrée de cette routine. La déclaration du thread provoque immédiatement son lancement et l'appel à la méthode join() permet d'attendre la fin du traitement.

```
void FnThread(vector<string> v)
{
  thread::id id = this_thread::get_id();
  for (string s : v) { cout << s << endl; }
}

void Use_Thread()
{
  vector<string> mes_filles = { «Edith», «Lisa», «Audrey» };
  thread t(FnThread, mes_filles);
  t.join();
  cout << «Thread finished !» << endl;
}
```

Il faut noter que l'implémentation des threads fournie par les dernières versions de Visual C++ utilise le Concurrency Runtime (ConCRT) alors que la librairie Boost tire partie des interfaces systèmes PThreads sous Linux et des API système Win32 sous Windows.

Méta programmation et type traits

Le fichier d'entête type_traits contient tout ce qu'il faut pour faire de la méta programmation. Ce sujet dépasse le cadre de cet article mais mérite d'être précisé.

C++14 et C++17

Le standard C++ évolue. C++14 sera une release mineure. C++17 va intégrer de nouvelles librairies dans la STL. Ces librairies sont inspirées de Boost comme FileSystem ou Network. Il y aura aussi un support des algorithmes parallèles.

Environnements de compilation C++11

Si vous êtes sous Linux ou Mac, vous avez la possibilité d'utiliser GCC ou CLang. Pour CLang, une version 3.x vous assure un maximum de confort. Pour GCC, une version 4.7 (sortie en mars 2012) est largement confortable. Il est possible de faire tourner GCC sous Windows via MinGW (http://mingw.org). Concernant Windows, Microsoft distribue Visual C++ dans la dernière version de Visual Studio 2013. Je vous conseille d'opter au minimum pour la version Visual Studio 2012 car la version Visual Studio 2010 n'offre qu'un support partiel. Il existe une version Visual Studio 2013 Express « for Desktop » qui est gratuite et qui tourne sous Windows 7 SP1 et plus.

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