Bien débuter avec Visual C++

Initialement, le développement sous Windows en C++ n'était possible qu'au travers d'un outil payant comme Visual C++ ou via l'acquisition de la suite Visual Studio. En plus de l'environnement de développement intégré (IDE), le package inclut les différentes en-têtes et librairies pour développer sous Windows alias Windows SDK, anciennement nommé Platform SDK.

> Windows SDK

Le Windows SDK fournit les outils, les compilateurs, les en-têtes et les librairies nécessaires à la création d'application sur les plate-formes Windows. Il existe aujourd'hui deux versions majeures de kits de développement « Windows SDK » : Windows SDK pour Windows 7 et NET Framework 3.5 SP1 ou Windows SDK pour Windows 7 et NET Framework 4.0. Les deux versions de Windows SDK sont disponibles en téléchargement gratuit. La première version redistribue le compilateur disponible dans Visual C++ 2008 SP1 (VC9) et la deuxième distribue le compilateur de Visual C++ 2010 SP1 (VC10). Pour lancer une compilation en mode de commande, il faut positionner quelques variables d'environnements comme PATH, INCLUDE et LIB. La définition exacte de l'environnement est décrite dans le fichier vcvars32.bat disponible sous %ProgramFiles%\Microsoft Visual Studio 10.0\VC\bin. Le kit Windows SDK fournit aussi un fichier qui permet de créer son environnement de build en utilisant SetEnv.cmd disponible sous %ProgramFiles%\Microsoft SDKs\Windows\v7.1\Bin et dont l'utilisation est la suivante :

```
Usage : Setenv [/Debug | /Release][/x86 | /x64 | /ia64 ][/vista
 | /xp | /2003 | /2008 | /win7][-h | /?]
```

Bien qu'il soit possible de compiler directement depuis la ligne de commande, l'utilisation d'un IDE apporte un plus grand confort de travail. Le plus simple est d'utiliser Visual C++ 2010 Express qui est gratuit et fournit le même environnement que Visual Studio 2010. VC++ Express distribue la runtime C, la librairie standard C++ (STL) et d'autres librairies comme PPL et les fichiers du Windows SDK. **Important** : Il faut installer Visual Studio 2010 SP1 après l'installation de Visual C++ 2010 Express. La version Express de Visual C++ 2010 possède quelques limitations comme l'absence des librairies MFC et ATL, pas de compilation 64 bit, pas de gestionnaire de fichiers de ressources (RC editor) pour désigner les interfaces graphiques, pas de support pour OpenMP ni pour les add-ins Visual Studio. Cependant, ces restrictions ne sont pas trop importantes pour quelqu'un qui veut démarrer dans le développement Visual C++. L'utilisation d'une bonne documentation est fondamentale. Il est possible de télécharger la documentation ou de l'utiliser en ligne. Je conseille pour ceux qui débutent une autre option qui est la documentation "MSDN Library pour Visual Studio 2008 SP1". Disponible gratuitement, cette image ISO de 2.2 GB qui contient la fameuse MSDN Library qui est la véritable Bible pour les programmeurs Windows. Pour bien démarrer, il suffit parfois de trouver un programme existant et de l'adapter. Les programmes d'exemples sont disponibles gratuitement. Pour tester notre environnement, tentons de compiler en ligne de commande sur un projet C++ sourceforge comme Scintilla, disponible sur http://sourceforge.net/projects/scintilla. Trouvons les deux makefiles et lançons la compilation via Microsoft Visual Studio 2010 Express -> Visual Studio Command Prompt.

```
D:\Files_GNU\scite300\scintilla\win32>nmake -f scintilla.mak
D:\Files_GNU\scite300\scite\win32>nmake -f scite.mak
```

Le fichier makefile contient les différentes commandes (rc, cl, ln) et options pour compiler la solution.

```
rc -fo.\ScintRes.res ScintRes.rc
cl -Zi -TP -W4 -EHsc -Zc:forScope -Zc:wchar_t -D_CRT_SECURE_
NO_DEPRECATE=1 -O1 -MT -DNDEBUG -GL -I../include -I../src -I..
/lexlib -c -Fo.\ ..\src\AutoComplete.cxx ..\src\CallTip.cxx ..
\src\CellBuffer.cxx …
link -OPT:REF -LTCG -DEBUG -DEF:Scintilla.def -DLL -OUT:..\bin
\Scintilla.dll .\AutoComplete.obj …
```

Si la compilation via le makefile est réussie, alors votre environnement Visual C++ est prêt à l'emploi.

> Introduction à C++

Visual C++ supporte le langage C++ et le langage C++/CLI. Le langage C++ standard ISO/IEC, dont la version C++11 vient d'être publiée par Herb Sutter et son équipe au mois de mars 2011 après de nombreuses années d'effort sur la standardisation. C'est le langage C++ standard, celui qui est disponible sur les environnements Windows, Linux et que l'on utilise avec un compilateur comme GCC ou Visual C++. Le deuxième langage C++ supporté est un langage C++/CLI standardisé ECMA qui comporte des extensions spécifiques pour produire du « code managé » à base de C++. Cet article ne traite que du langage C++ standard et n'adresse pas C++/CLI. Dans certains articles, revues ou webcasts, on peut entendre que C++ est un langage « unsafe », non sécurisé et dangereux. Ceci est faux. C'est une ruse pour favoriser l'adoption d'autres langages. C++ est un langage type-safe, qui fournit des mécanismes d'abstractions évolués (classe, template) et qui ne sacrifie pas le pouvoir et les performances.

La Runtime C

Le langage dispose d'une librairie nommée Runtime C qui contient les routines de base pour réaliser des applications. Cette runtime est implémentée en langage C et fournit les fonctions communément définies dans le C ANSI, qui fait l'objet du livre K&R « C ANSI ». Avec Visual C++, le code source de la runtime est fourni dans le répertoire %ProgramFiles%\Microsoft Visual Studio 10.0\VC\crt\src. Dans Visual C++ 2010, la CRT contient 1300 fichiers pour 15 MB de source code. La runtime C fournit les catégories de fonctions suivantes : tableau 1.

La librairie standard C++ (STL)

La libraire STL est constituée de classes et de templates définis en 6 parties : containers, container adapters, iterators, algorithms, function objects et function adapters. Les éléments de la famille « containers » sont équivalents aux collections. Ces classes permettent de stocker,

supprimer ou d'accéder à certains éléments. Voici la liste des containers : tableau 2. Les iterators ou itérateurs en français permettent l'accès aux containers. Ils fonctionnent comme des pointeurs et peuvent être de différents styles : input, output, forward, bidirectional, random access. Chaque container utilise un type d'itérateur spécifique. La documentation associée à chaque classe indique lequel utiliser. Un exemple illustre comment utiliser les containers et les itérateurs.

```
#include <vector>
#include <string>
#include <iostream>
using namespace std;
…
```

Tableau 1

Catégorie	Fonctions C	Fichier d'en-têtes
gestion des arguments	va_arg, va_end, va_start	stdarg.h
manipulation de buffer	memcpy, memchr, memcmp, memmove, memset, swab	ctype.h
classification de byte et caracter	isalpha, isalnum, ismbbkalnum, ismbbkprint, isdigit, islower, isupper, isprint, ispunct, isspace, isxdigit…	ctype.h, wchar.h
alignement de données	aligned_malloc, aligned_free, aligned_msize, aligned_realloc…	malloc.h
conversion de données	abs, atof, atoi, atol, ecvt, fcvt, itoa, mbtowc, strtod, strtol, toascii, tolower, toupper…	stdlib.h, math.h
routines de debugging	_ASSERT, _ASSERTE, _CrtDbgBreak, _CrtDbgReport, _CrtIsValidPointer, malloc_bdg, free_dbg, realloc_dbg…	crtdbg.h
contrôle de répertoire	chdir, getcwd, getdrive, getdiskfree, mkdir, rmdir, searchenv…	direct.h, errno.h
gestion des erreurs CRT	assert, clearerr, _RPT, set_error_mode…	crtdbg.h
gestion des exceptions et robustesse	terminate, unexpected, set_terminate, set_unexpected…	eh.h
gestion des fichiers	chsize, filelength, fstat, get_osfhandle, locking, setmode, access, chmod, fullpath, makepath, mktemp, remove, rename, splitpath, umask, unlink…	io.h, errno.h
support des flottants	abs, acos, asin, atan, atof, cos, div, floor, fpclass, frexp, hypot, max, min, mod, pow, rand, sin, sqrt, srand, tan…	stdlib.h, math.h
entrées et sorties	Stream I/O: fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, fopen, fprintf, fputc, fputs, fread, fscanf, fseek, fwrite, setbuf… Low-level I/O: close, commit, creat, dup, eof, lseek, open, read, sopeb, tell, umask, write… Console I/O: gets, printf , puts, scanf, getch...	stdio.h, io.h, sys/stat.h, sys/types.h
internationalisation et unicode	setlocale, _UNICODE; _MBCS…	locale.h
gestion de la mémoire	calloc, malloc, delete, expand, msize, realloc...	stdlib.h, malloc.h
processus et environnement	abort, atexit, _beginthreadex, cwait, execv, execvpe, getenv, getpid, _pipe, _popen, raise, signal, system…	process.h, errno.h
recherche et tri	bsearch, lfind, lsearch, qsort…	stdlib.h, search.h
manipulation de chaînes	strcol, strdec, strinc strncnt, strspnp, strcat, scanf, sprintf, strchr, strcmp, strcpy, strdup, strerror, strftime, strlen, strlwr, strcat, strspn, strstr, strtok, strupr…	string.h
appel système	findclose, findfirst, findnext…	io.h
gestion du temps	asctime, clock, ctime, difftime, gmtime, localtime, mktime, strdate, strftime, time…	time.h

```
void routine2()
{
  vector<string> vector1;
  // Ajout d'éléments
  vector1.push_back(«C++ Renaissance»);
  vector1.push_back(«Going Native»);
  vector1.push_back(«Why C++ ?»);
  // Obtention de l'itérateur -> vector<string>::iterator it =
vector1.begin()
  // syntaxe C++ 11 -> auto it = vector1.begin();
  for(vector<string>::iterator it = vector1.begin() ; it!=vector1
.end() ; it++)
  {
    cout << «value: « << *it << endl;
  }
  // Itération dans une boucle for classique
  for(size_t i=0 ; i<vector1.size() ; i++ )
  {
    cout << «value: « << vector1[i] << endl;
  }
}
```

Pour ajouter des éléments dans un container vector<T>, on utilise la méthode push_back(T). Pour itérer dans le container, on obtient l'itérateur propre au container, puis on itère en utilisant l'opérateur ++. L'accès à l'élément se fait en utilisant l'opérateur *. Une itération peut aussi se faire via l'utilisation du vector<T> comme un tableau en utilisant l'opérateur []. La STL fournit plusieurs moyens pour réaliser les opérations et ainsi permet plusieurs styles d'écriture de code. Il existe un container très sympa à utiliser qui permet d'associer une clé avec une valeur : map<K,T>. L'ajout d'éléments se fait soit en utilisant l'opérateur [] ou bien la méthode insert qui prend un type pair<first, second>. L'itération dans le container se fait en utilisant les membres first et second. La recherche se fait en utilisant la méthode count(). Exemple de code :

Tableau 2

Container	Fichier d'en têtes	Description
vector	<vector>	vector<T> est un tableau qui grossit automatiquement. C'est le container le plus utilisé
deque	<deque>	deque<T> est une double-ended queue. C'est comme un vector mais plus rapide pour l'insertion en début ou en fin
list	<list>	list<T> permet l'ajout et la suppression rapide d'éléments
map	<map>	map<K,T> stocke des couples key/types uniques
multimap	<map>	multimap<K,T> permet que les couples soient identiques
set	<set>	set<T> est un ensemble d'éléments uniques
multiset	<set>	multiset<T> permet que les éléments soient identiques
hash_map	<hash_map>	hash_map<K, T> est un map rapide en accès
hash_multimap	<hash_map>	hash_multimap<K, T> est un multimap rapide en accès
hash_set	<hash_set>	hash_set<T> est un set rapide en accès
hash_multiset	<hash_set>	hash_multiset<T> est un multiset rapide en accès
stack	<stack>	stack<T> implémente LIFO (last in, first out) donc dernier entré, premier sorti
queue	<queue>	queue<T> implémente FIFO(first in, first out) donc premier entré, premier sorti
priority_queue	<queue>	priority_queue<T> est une queue dans laquelle l'élément le plus haut est le premier dans la queue

```
#include <map>
…
void routine3()
{
 map<string, string> dico;
 dico[«VB»] = «Visual Basic»;
 dico[«C#»] = «Visual C# -> Productivity»;
 dico[«C++»] = «Visual C++ -> Power and Performance !»;
 // Ajout en utilisant la fonction make_pair
 dico.insert(make_pair(«C++11», «Fast and Fluid ! See you later
 in Windows 8»));
 // Obtention de l'itérateur via container<T>::iterator
 for( map<string, string>::iterator it = dico.begin() ; it!=
dico.end() ; it++)
 {
   cout << «key: « << it->first << « value:» << it->second << endl;
 }
 // count détermine si une clé existe -> retourne 0 ou 1
 if(dico.count(«C++») != 0)
 {
   cout << «C++ value: « << dico[«C++»] << endl;
 }
}
```

Les containers de la STL sont génériques, donc on peut les utiliser avec n'importe quel type. Exemple de map avec comme clé un type string et comme valeur un type LangageProp :

```
class LangageProp
{
private:
 string desc, core_value;
public:
 LangageProp(string d, string cv ) : desc(d), core_value(cv) {}
 string Display() { return desc + « is « + core_value; }
};

void routine4()
{
 map<string, LangageProp> dico;
 dico.insert(make_pair(«VB», LangageProp(«Visual Basic», «for the
 kids ?»)));
 dico.insert(make_pair(«C#», LangageProp(«Visual C#», «for
 productivity»)));
 dico.insert(make_pair(«C++», LangageProp(«Visual C++», «for Power
 and Performance !»)));
 // Obtention de l'itérateur via container<T>::iterator
 for( map<string, LangageProp>::iterator it = dico.begin() ; it!
=dico.end() ; it++)
 {
   cout << «key: « << it->first << « object:» << it->second.
Display() << endl;
 }
}
```

Dans cette exemple, on remarquera que le membre second de l'itérateur est de type LangageProp et on peut faire appel directement à la méthode Display() de cette classe. La STL fournit aussi un ensemble de fonctions templates appelées algorithmes disponibles dans le fichier d'en-têtes <algorithm>. Ce fichier contient plusieurs dizaines de fonctions génériques qui savent traverser les containers et y opérer un certain traitement. La plus connue est sans doute for_each qui permet de faire un même traitement pour l'ensemble des éléments d'un container. Nous reprenons comme base la collection ci-dessus. L'appel à for_each se fait sur la base d'un itérateur de début, d'un itérateur de fin et d'une routine, OnItem, qui va être appelée sur chaque élément du container.

```
void OnItem(pair<string, LangageProp> value)
{
 // retrieve the LangageProp object reference
 LangageProp &prop = value.second;
 cout << «OnItem...» << prop.Display() << endl;
}

void routine5()
{
 map<string, LangageProp> dico;
 dico.insert(make_pair(«VB», LangageProp(«Visual Basic», «for the
 kids ?»)));
 dico.insert(make_pair(«C#», LangageProp(«Visual C#», «for
 productivity»)));
 dico.insert(make_pair(«C++», LangageProp(«Visual C++», «for
 Power and Performance !»)));
 for_each(dico.begin(), dico.end(), OnItem);
}
```

Programmation moderne avec C++11

Écrire du C++ moderne ça veut dire quoi ? Par exemple, on ne va plus explicitement faire de simples new/delete sur nos objets. On va construire et manipuler les objets avec des « smart pointer » via shared_ptr<T>. Ce template implémente un mécanisme de compteur de références qui permet de partager des objets entre plusieurs clients, et permet de stocker des pointeurs dans des containers en toute sécurité.

```
#include <memory>
…
void routine6()
{
 shared_ptr<LangageProp> lp1(new LangageProp(«C++11», «writing
 modern C++ code»));
 cout << lp1->Display() << endl;
 shared_ptr<LangageProp> lp2(lp1);
 cout << «use_count should be 2. use_count() ==» << lp2.use
_count() << endl;
}
```

L'abstraction weak_ptr<T> est le meilleur compagnon de shared_ptr<T>. Cela permet de casser les références circulaires. Le template weak_ptr<T> implémente la pattern Observer pour les shared_ptr<T>. Avec un weak_ptr<T>, dès qu'un shared_ptr<T> libère ses ressources, l'information est propagée à tous les weak_ptr<T> afin qu'il ne puisse plus utiliser un pointeur invalide.

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