Je pratique le C++

Partie 4/5

Level 300

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 la librairie standard du C++ nommé STL : Standard Template Library.

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

Le langage C possède son runtime et le C++ possède aussi son runtime, appelé STL. Dans le monde Microsoft, le runtime du C se nomme MSVCRTxxx.dll et celui du C++ se nomme MSVCPxxx.dll. Cependant, la STL ne se comporte pas comme une liste de fonctions que l'on peut appeler – exemple fopen en C.

La STL est constituée de templates. SI vous avez lu l'article précédent, nous y avons étudié les templates en douceur. Cela veut dire plusieurs choses. La STL est organisée via des fichiers d'entêtes.

Les fonctionnalités offertes par la bibliothèque standard peuvent être classées comme suit :

- Support du runtime du langage (par exemple, pour l'allocation et les informations de type runtime) ;
- La bibliothèque C standard (avec des modifications vraiment mineures pour minimiser la violation du système de type) ;
- Les chaînes (avec le support des jeux de caractères internationaux et localisation) ;
- Support pour la correspondance des expressions régulières ;
- Les flux d'entrée/sortie sont un framework extensible pour l'entrée et la sortie vers lequel les utilisateurs peuvent ajouter leurs propres types, flux, stratégies de buffering, locales et jeux de caractères ;
- Un framework de conteneurs (comme vector et map) et d'algorithmes (comme find(), sort(), et merge()). Ce framework, appelé conventionnellement la STL, est extensible pour que les utilisateurs puissent y ajouter leurs propres conteneurs et algorithmes ;
- Support pour le calcul numérique (comme les fonctions mathématique standard, les nombres complexes, les vecteurs avec des opérations arithmétiques, et les générateurs de nombres aléatoires) ;
- Support pour la programmation parallèle, incluant les threads et les verrous. Le support parallèle est fondamental pour que les utilisateurs puissent ajouter le support aux nouveaux modèles parallèles dans des bibliothèques ;
- Des classes utilitaires pour le support de la méta programmation à base de templates (par exemple, type traits, la programmation générique avec le style STL (par exemple, pair), et la programmation générale (par exemple, clock) ;
- Les ''Pointeurs Intelligents'' pour la gestion des ressources (par exemple, unique_ptr et shared_ptr) et une interface pour la libération des ressources ;
- Des conteneurs spéciaux, comme array, bitset, et tuple.

Les critères principaaux pour inclure une classe dans la bibliothèque étaient que :

- Cela pourrait être utile à presque tous les programmeurs C++ (les novices ou les experts) ;
- Cela pourrait être fournie sous une forme générale qui n'ajoute pas de surcharge importante par rapport à une version simplifiée de la même fonctionnalité ;
- Que l'utilisation doit être simple à apprendre (par rapport à la complexité de la tâche).

La bibliothèque standard de C++ fournit essentiellement les structures de données fondamentales les plus courantes ainsi que les algorithmes fondamentaux à utiliser avec. Nous allons balayer toute ou partie de cette liste exhaustive.

Sélection des fichiers d'entêtes de la bibliothèque standard:	
<algorithm>	copy(), find(), sort()
<array>	array
<chrono>	duration, time_point
<cmath>	sqrt(), pow()
<complex>	complex, sqrt(), pow()
<forward_list>	forward_list
<fstream>	fstream, ifstream, ofstream
<future>	future, promise
<ios>	hex,dec,scientific,fixed,defaultfloat
<iostream>	istream, ostream, cin, cout
<map>	map, multimap
<memory>	unique_ptr, shared_ptr, allocator
<random>	default_random_engine, normal_distribution
<regex>	regex, smatch
<string>	string, basic_string
<set>	set, multiset
<sstream>	istrstream, ostrstream
<stdexcept>	length_error, out_of_rang e, runtime_error
<thread>	thread
<unordered_map>	unordered_map, unordered_multimap
<utility>	move(), swap(), pair
<vector>	vector

Le type string

Ce type est défini dans le fichier d'entête <string> et c'est le type qui permet de faire la liaison entre les type char du C et le type string d'un niveau plus haut qu'est le type string.

Le type string permet de concaténer des strings entre elles. Ce n'est pas un pointeur de char. C'est un template.

```
typedef basic_string<char, char_traits<char>, allocator<char> >
  string;
```

Il est possible de fournir son propre allocateur mémoire. Mais qui va utiliser cela ? Et pourtant c'est prévu... Rappelez-vous dans l'article sur les classes lorsque je parlais de la casquette auteur de classe et de la casquette utilisateur de classe. On y est exactement sauf que c'est une classe template.

```
string msg = "Le nutty est coquine !";
string msg2 = "Maggie est la reine des coquines !";
string s = msg + " " + msg2;
cout << s << endl;
string nutty = s.substr(0, 9);
cout << nutty << endl;
msg[3] = toupper(msg[3]);
cout << msg << endl;
string temp("VS 2015 is here !");
```

```
size_t t = temp.find_first_of("here");
cout << temp << " " << t << endl;
```

La classe string contient de nombreuses opérations qui évitent le parcours des chaînes C à l'ancienne. Pour passer d'une string à un type C char *, il faut utiliser la fonction c_str() :

```
const char * psz = msg.c_str();
printf("String const char * = %s\n", psz);
```

Les expressions régulières (regex)

La fonctionnalité des regex est disponible dans le fichier d'entête <regex>. Le support est le même que dans d'autres langages ; on y trouve les fonctions regex_match, regex_search, regex_replace, regex_iterator, regex_token_ietrator.

Les IO/Streams

La STL utilise les flux pour gérer des I/O avec des buffers. La fonction la plus connue est cout et son opérateur '<<'.

```
cout << "On n'est pas que des Mickey !" << endl;
int i = 10;
float f = 2.5;
cout << i << ", " << f << endl;
```

Le meilleur ami de cout est cin. Cela permet de capter les entrées du clavier :

```
int j = 0;
cout << "Give ma a Int !" << endl;
cin >> j;
cout << "Merci pour " << j << endl;
```

En plus de gérer les types standards et les strings, il est possible de tirer parti de la librairie pour afficher d'autres types. Exemple, considérons le type suivant :

```
struct CPersonne
{
  int age;
  string name;
};

ostream& operator<<(ostream& os, const CPersonne& p)
{
  return os << p.name << " " << p.age << " ans";
}

void TestIO()
{
  CPersonne p;0
  p.age = 5;
  p.name = "Audrey Maggie";

  cout << "La soeur de Lisa c'est " << p << endl;
}
```

La définition de l'opérateur << avec le type ostream permet de passer un type CPersonne à cout et cela sans forcer ! Autres aspects de la librairie I/O, c'est le formatage.
Pour afficher des types comme le fait une fonction printf en C avec les %d, %f, %x et %s ont leur équivalent dans la librairie.

```
void TestFormating()
{
  float f = 2.50;
  cout << f << ";"
     << scientific << f << ";"
     << hexfloat << f << ";"
     << fixed << f << ";"
     << defaultfloat << f << endl;
}
```

La gestion des fichiers est assurée au travers du fichier d'entêtes <fstream> :

- ifstream permet de lire un fichier,
- ofstream permet d'écrire un fichier,
- fstream permet de lire et d'écrire dans un fichier.

Exemple d'écriture de fichier :

```
ofstream ofs("c:\\temp\\MyGirls.txt");
ofs << "Edith" << endl;
ofs << "Lisa" << endl;
ofs << "Maggie" << endl;
ofs.close();
```

C'est vraiment très simple. La librairie I/O permet de gérer les buffers. Le fichier d'entête <sstream> permet de manipuler des chaînes :

- istringstream pour lire des chaînes,
- ostringstream pour écrire des chaînes,
- stringstream pour lire et écrire des chaînes.

```
int i = 20;
float f = 5.75;
string s = "Maggie est trop coquine !";
ostringstream oss;
oss << i << ", " << f << ", " << s;

string str = oss.str();
cout << str << endl;
```

Les containers

Le container le plus utilisé est vector<T>. Il est disponible dans le fichier d'entête <vector>. C'est une séquence d'éléments d'un type donné. Les éléments sont stockés de manière contiguë. Pour ajouter des éléments à un vector, il faut utiliser la méthode push_back().
Le parcours d'un vector peut se faire avec un range-for ou bien en utilisant un itérateur. Les containers de la STL sont tous accessibles au travers un itérateur. Les fonctions begin(), end(), operator++, operator–m operator* permettent de manipuler un itérateur. Exemple :

```
vector<CPersonne> v = { {12, "Edith"}, {9, "Lisa"}, {5, "Maggie"} };
CPersonne p;
p.age = 41;
p.name = "Papa";
v.push_back(p);
for (auto item : v)
{
   cout << item.age << " " << item.name << endl;
}
for (vector<CPersonne>::const_iterator it = begin(v); it != end(v); ++it)
{
   CPersonne p = *it;
   cout << p.age << " " << p.name << endl;
}
```

Voici la liste des opérations principales pour vector<T> :

Opération	Explication
v.empty()	Retourne true si v est vide. Sinon retourne false.
v.size()	Retourne le nombre d'élements dans v.
v.push_back(t)	Ajoute un élément de valeur t à la fin de v.
v[n]	Retourne une référence vers l'élément en position n dans v.
v1=v2	Remplace les éléments dans v1 avec une copie des éléments dans v2.
v1={a, b, c...}	Remplace les éléments dans v1 avec une copie des éléments de la liste.
v1==v2	v1 et v2 sont égaux s'il ya le même nombre d'él éments et de valeur.
v1!=v2	opposé de v1==v2
<, <=, >, >=	Suivant l'ordre des valeurs retourne un bool

Il existe un container qui représente une double liste chaînée au travers de list. Il est disponible au travers le fichier d'entête <list>.

```
list<CPersonne> l = { { 12, "Edith" },{ 9, "Lisa" },{ 5, "Maggie" } };
CPersonne p = { 41, "Papa" };
l.push_front(p);
CPersonne p2 = { 40, "Maman" };
l.push_back(p2);
for (auto item : l)
{
    cout << item.age << " " << item.name << endl;
}
```

Le container map<K,V> est très utile. Il est disponible dans le fichier d'entête <map>. C'est un container associatif.

```
map<string, int> family = { { "Edith", 12 },{ "Lisa", 9 },{ "Maggie", 5 } };
family["Papa"] = 41;
for (map<string, int>::const_iterator it = begin(family); it != end(family); ++it)
{
    string name = it->first;
    int age = it->second;
    cout << name << " " << age << endl;
}
```

Il existe d'autres containers dans la STL comme la hashtable nommée unordered_map. Le container hashtable et ses dérivées (voir tableau ci-dessous) ne contiennent pas le terme hashtable pour des soucis de nommage. Il y a surement du code existant qui a fait une classe ou un template hashtable, et c'est pour éviter la collision de nom.

Les containers disponibles dans la STL	
vector<T>	un vecteur de taille variable
list<T>	une liste doublement chaînée
forward_list<T>	une liste chaînée
deque<T>	une queue
set<T>	un set (une map avec une clé sans valeur)
multiset<T>	un set qui peut être en doublon
map<K,V>	un tableau associatif
multimap<K,V>	une map avec une clé qui peut être en doublon
unordered_map<K,V>	une map qui utilise un lookup hashtable
unordered_multimap<K,V>	une multimap qui utilise un lookup hashtable
unordered_set<K,V>	un set qui utilise un lookup hashtable
unordered_multiset<K,V>	un multiset qui utilise un lookup hashtable

Les algorithmes

La STL fournit des fonctions simples pour parcourir des ensembles, faire des copies, des insertions, des suppressions, des recherches simple ou complexes. Le fichier d'entête est <algorithm>. La force des algorithmes réside dans le fait qu'ils prennent pour la plupart un itérateur de début et un itérateur de fin afin de réaliser le parcours sur un ensemble fini. C'est un peu déroutant au début et puis finalement, on s'aperçoit que la plupart des parcours proposés dans ce fichier d'entête sont bien faits. Il faut maîtriser les itérateurs : c'est la seule contrainte. Dans le tableau ci-dessous, b vaut begin() et e vaut end().

Sélection d'algorithmes dans la STL	
p=find(b,e ,x)	p est le premier p dans [b:e) de telle manière que *p==x
p=find_if(b,e ,f)	p est le premier p dans [b:e) de telle manière que f(*p)==true
n=count(b,e ,x)	n est le nombre d'éléments *q dans [b:e) de telle manière que *q==x
n=count_if(b,e ,f)	n est le nombre d'éléments *q dans [b:e) de telle manière que f(*q,x)
replace(b,e ,v,v2)	Remplace les éléments *q dans [b:e) de telle manière que *q==v par v2
replace_if(b,e ,f,v2)	Remplace les éléments *q dans [b:e) de telle manière que f(*q) par v2
p=copy(b,e ,out)	Copie [b:e) dans [out:p)
p=copy_if(b,e ,out,f)	Copie les éléments *q de [b:e) de telle manière que f(*q) jusqu'a [out:p)
p=move(b,e ,out)	Déplace [b:e) vers [out:p)
p=unique_copy(b,e ,out)	Copie [b:e) vers [out:p); ne copie pas les duplicates adjacents
sort(b,e)	Trie les éléments de [b:e) en utilisant < comme critère de tri
sort(b,e,f)	Trie les éléments de [b:e) en utilisant la fonction de tri f
(p1,p2)=equal_range(b,e ,v)	[p1:p2) est la subséquence de tri [b:e) avec la valeur v; un binary search de v
p=merge(b,e ,b2,e2,out)	Merge deux séquences [b:e) et [b2:e2) dans [out:p)

Exemple avec find :

```
vector<string> v = { "Edith", "Lisa", "Maggie" };

//auto res = find(begin(v), end(v), "Maggie");
vector<string>::iterator res = find(begin(v), end(v), "Maggie");

if (res == end(v))
{
    cout << "Not found !" << endl;
}
else
{
    cout << "Found ! " << *res << endl;
}
```

Les templates utilities

Tout dans la STL n'est pas aussi simple que les containers ou la librairie I/O stream. Il existe des classes templates qui permettent de tirer parti de fonctionnalités avancées. Considérons la gestion des ressources, il existe deux templates qui sont unique_ptr<T> et share_ptr<T>. unique_ptr<T> représente la possession unique. shared_ptr<T> représente la possession partagée. Disponible dans le fichier d'entête <memory>, ces « smart pointers » ou pointeurs intelligents permettent de ne plus coder le delete, c'est le template qui s'en occupe. Le principal avantage d'utiliser les smart pointers est d'éviter les fuites mémoire.

```
unique_ptr<CPersonne> ptr(new CPersonne());
ptr->age = 41;
ptr->name = "Itchy";
```

```
// use ptr
// delete fait automatiquement
```

Voici la liste des opérations communes entre unique_ptr<T> et shared_ptr<T> :

Opération	Explication
shared_ptr<T> sp	Smart pointer null qui pointe sur un objet T
unique_ptr<T> up	Smart pointer null qui pointe sur un objet T
p	Utilise p comme une condition; true si p pointe sur un objet
*p	Déréférence p pour obtenir l'objet sur lequel p pointe
p->mem	Synonyme pour (*p).mem
p.get()	Retourne le pointeur dans p.
swap(p,q)	Swap les pointeurs dans p et q
p.swap(q)	Swap les pointeurs dans p et q

Le template share_ptr<T> possède quelques subtilités :

Opération	Explication
make_shared<T>(args)	Retourne un shared_ptr sur la mémoire allouée et initialise l'objet via args
shared_ptr<T> p(q)	p est une copie de q. Incrémente le compteur de référence interne
p=q	Incrémente le compteur de référence de q
p.use_count()	Retourne le nombre d'objets partagés avec p
p.unique()	Retourne true si p.use_count vaut 1 sinon false

Le template array<T, C> permet de gérer les tableaux aussi rapidement que les built-in arrays.

```
array<string, 3> ar;
ar[0] = "Edith";
ar[1] = "Lisa";
ar[2] = "Audrey";
for (auto element : ar)
{
    cout << element << endl;
}
```

Le template pair<T, U> représente deux éléments et est disponible dans le fichier d'entête <utility>. Utilisez make_pair pour remplir l'objet pair.

```
pair<string, float> p;
p.first = "The C++ Object Model";
p.second = 50.0;
cout << p.first << " " << p.second << endl;
pair<string, string> p2 = make_pair("Maggy", "t'es une coquine !");
cout << p2.first << " " << p2.second << endl;
```

Le template tuple<T...> représente une séquence de types variés et de types différents. Utilisez make_tuple pour remplir l'object tuple.

```
tuple<string, float, string> t;
t = make_tuple("C++ Primer", 50.0, "The best of all books !");
cout << get<0>(t) << ";" << get<1>(t) << ";" << get<2>(t) << endl;
```

Concurrency : le multithreading !

Il est possible de lancer une tâche en parallèle et d'en attendre la fin. On va utiliser la classe thread disponible dans le fichier d'entête <thread>. Il suffit de passer une routine en argument du constructeur de la classe thread. La méthode join() sur l'objet thread permet d'en attendre la fin.

```
void ThreadFunc()
{
  // ../..
}

void TestThread()
{
  thread t(ThreadFunc);
  std:thread::id id = t.get_id();
  t.join();
  cout << "TestThread: TID:" << id << ", " << "Main:" << GetCurrentThreadId() << endl;
}
```

Dans l'exemple ci-dessus, la fonction du thread ne prend pas de paramètres. Cependant, dans certains cas, on veut pouvoir passer des paramètres au thread. Il suffit de passer les paramètres au constructeur de l'objet thread :

```
class Param
{
public:
  string s;
  int i;
  float f;
};

void ThreadFunc2(Param param)
{
  cout << param.s << ", " << param.i << ", " << param.f << endl;
  // ../..
}

void TestThreadWithParam()
{
  Param param = { "C++", 14, 50.0 };
  thread t(ThreadFunc2, param);
  t.join();
}
```

Il existe des classes de type verrou comme mutex ou unique_lock<T> pour protéger l'accès aux données partagées. Vous pouvez remarquer que la gestion des threads est plutôt simple à utiliser.

Conclusion

Nous avons balayé les différentes composantes de la librairie standard et le constat est le suivant : le code source de la STL est complexe car c'est un framework extensible ; on l'a vu sur iostream et le passage d'une structure à cout ; qu'il faut apprendre à maîtriser. A partir des exemples simples présents dans cet article, vous n'avez plus qu'à vous lancer. Il y a des domaines que je n'ai pas couvert et c'est volontaire mais sachez que cet article couvre 90% de la STL. L'aspect le plus important est la maîtrise des containers. Ne construisez plus vos propres structures de list, hashtable ou autres, utilisez la STL ! La seule utilisation de vector<T> vous fait rentrer de plein pied dans la STL. Vous aurez des performances inégalables. N'utilisez plus les delete et passez aux smart pointers ! unique_ptr<T> et shared_ptr<T> justifient ; comme vector<T> ; une utilisation systématique. La fin des memory leak (fuite mémoire) en est une autre justification. Un conseil : laissez le code un peu ancien (legacy) avec un style à la papa, et sur le nouveau code, utilisez la STL. Le nouveau code doit être codé avec tous les artifices du C++ 11. Le code est plus lisible, les performances sont au rendez-vous. « Power & performance » comme disait Herb Sutter, chairman du standard C++ ISO. ⊞