Je pratique le C++

Partie 1/5

Nous vous proposons une série d'articles sur la pratique de C++ pour que vous puissiez tous vous y mettre. Pas besoin de débourser un centime pour pratiquer car tout est gratuit. Que ce soit l'ancien Visual Studio Express, Visual Studio Community Edition 2013 ou GCC, ou CLang via Linux ou MinGW, ça ne coûte rien. L'essentiel est d'avoir un compilateur récent pour bénéficier du C++ 11 alias le C++ Moderne. C++ 11 apporte énormément de changement à C++ 03 et C++05TR1, et c'est la raison pour laquelle il y a un engouement énorme autour de C++ 11. C'est la renaissance de C++. C++ est le langage incontournable pour faire des applications sur téléphone Android, iOS, Windows Phone, mais aussi pour faire des applications sur Windows, Linux et Mac. Et pourtant on n'en fait pas beaucoup de publicité.

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

C++ est un langage normalisé par un comité ISO qui améliore constamment le langage et la librairie standard STL (Standard Template Library). Le comité ISO est en préparation de la version C++17 (qui fait suite aux versions C++11, C++14).

Les types prédéfinis

Il existe de nombreux types prédéfinis :

Type	Sens	Taille minimum
bool	Booléen. true ou false.	NA
char	Charactère	8 bits
wchar_t	Charactère large	16 bits
char16_t	Charactère Unicode	16 bits
char32_t	Charactère Unicode	32 bits
short	Entier court	16 bits
Int	Entier	16 bits
long	Entier long	32 bits
long long	Entier long	64 bits
float	Flottant à précision simple	6 digits
double	Flottant à double précision	10 digits
long double	Flottant à double précision	10 digits

Les types peuvent être signés ou non via le specifier signed ou unsigned. Les types sont tous signés par défaut. La taille d'une variable peut être obtenue via sizeof(t).

Les variables

Une variable se déclare comme suit :
<type> nom_de_variable = valeur_d'initialisation ;
L'initialisation n'est pas obligatoire mais si elle est manquante, le contenu de la variable est indéfini. En mode DEBUG, ça vaudra 0 ou null mais en RELEASE c'est indéfini. Conseil : il faut initialiser ses variables.

Les références

Une référence pointe sur un objet. C'est une référence vers quelque chose. C'est un alias.

```
void Sample_References()
{
   int a = 10;
   int &b = a;
   a++;
   cout << "b=" << b << endl;
}
```

L'affichage sera : b=11. Les références sont comme des pointeurs mais avec un pouvoir limité. (ça gratte ?).

Les pointeurs

Un pointeur sert à désigner un espace mémoire qui pointe vers quelque chose, un objet ou une variable. C'est comme une référence sauf qu'il n'est pas obligé d'être initialisé à la déclaration.

```
void Sample_Pointer()
{
   int a = 10;
   int * ptr = nullptr;
   ptr = &a;
   a++;
   cout << "ptr adress=" << ptr << " value=" << *ptr << endl;
}
```

L'affichage sera : ptr adress=0x0062FE20 value=11
Dans cet exemple, on initialise le pointeur à nullptr. Ensuite on indique que ptr vaut l'adresse (via &) de a. La valeur de a est son emplacement mémoire et le contenu du pointeur est accessible via *.

const

Utiliser const sur une variable empêche cette variable d'être modifiée. Cependant, elle doit être initialisée à la déclaration.

```
void Sample_Const()
{
   const int a = 100;
   a++;    // error
   a = 200;// error
}
```

Par défaut, les objets const sont locaux à un fichier .cpp pour la compilation. Pour définir et partager un objet const à plusieurs, il faut déclarer cette variable comme extern sinon le compilateur indiquera ne pas trouver la variable avec un message du style « var is undefined. ».

Références const

Il existe aussi des références const. Ce sont des références que l'on ne peut pas changer.

```
void Sample_ConstRef()
{
   int a = 10;
   const int & aref = a;
   int b = 20;
   aref = b; // error aref est const
}
```

Alias de types

Il est possible de définir un alias sur un type. Pourquoi ? C'est pour rendre le programme plus lisible.

```
typedef std::string String;
typedef vector<string> MesFilles;
typedef vector<string>::iterator IT;

void Sample_Typedef()
{
   String maggie = "je m'appelle Maggie et j'ai bientôt 5 ans et j'aime Papa";
   cout << maggie << endl;

   MesFilles girls;
   girls.push_back("Edith");
   girls.push_back("Lisa");
   girls.push_back("Audrey Maggie");
   for (IT it = begin(girls); it != end(girls); ++it)
   {
      cout << *it << endl;
   }
}
```

Dans cet exemple, j'ai créé 3 alias : sur std ::string, sur le vector<string> et sur son iterator.

Le mot-clé auto

Disponible depuis C++ 11, le mot clé auto permet de laisser le compilateur déduire le type d'un objet. C'est le var du C# pour ceux qui connaissent.

```
void Sample_auto()
{
   MesFilles girls = { "Edith", "Lisa", "Audrey" };
   for (auto it = begin(girls); it != end(girls); ++it)
   {
      cout << *it << endl;
   }
}
```

Vous remarquez que j'ai initialisé le vector de string (typedef MesFilles) avec des {}. C'est le C++ 11 qui permet cela.

Les structures

Pour créer un type utilisateur qui contient des variables, le plus simple c'est de créer une structure.

```
struct Book
{
   string name;
   int price;
   string comments;
};

void Sample_Book()
{
   Book CPPPrimer5ThEdition = { "C++ Primer", 50, "the bible" };
   cout << "book :" << CPPPrimer5ThEdition.name << endl;
   cout << "price : " << CPPPrimer5ThEdition.price << endl;
   cout << "comments : " << CPPPrimer5ThEdition.comments << endl;
}
```

Dans cet exemple, la structure est déclarée en dehors de la fonction et la définition d'une variable de ce type se fait avec les {}. On aurait pu aussi initialiser les membres un par un.

```
   Book ref;
   ref.name = "The C++ language";
   ref.price = 50;
   ref.comments = "by Bjarne Stroustrup, the creator of C++";
```

Une structure est comme une classe dont tous les membres sont "public" alors qu'ils sont "private" pour une classe.

Les entêtes .h (ou .hpp)

Un programme peut contenir plusieurs fichiers .cpp. Et il est nécessaire de partager des informations sachant que lorsque le compilateur prend un fichier, il faut que toutes les déclarations de types soient connues. Donc, on écrit des fichiers header (.h) qui contiennent toutes déclarations que nous allons utiliser. Si je veux que la structure Book soit accessible par plusieurs fichiers .cpp, il faut que je la définisse dans un header .h et que je fasse un #include au début du fichier .cpp qui veut l'utiliser pour que le compilateur puisse faire son job. Voici donc la version « propre » du sample qui utilise la structure book :

```
#include "Book.h"

void Sample_Book()
{
   Book CPPPrimer5ThEdition = { "C++ Primer", 50, "the bible" };
   ...
```

Les entêtes précompilées

Par convention, on met les #include au début du fichier .cpp. Pour optimiser le temps de compilation, il existe quelque chose qui se nomme les entêtes précompilées. Il suffit de mettre tous ses fichiers d'entêtes dans un seul header (stdafx.h pour le Visual C++) et de paramétrer le projet pour dire que ces entêtes doivent être précompilées ainsi, le fichier stdafx.cpp contient le #include stdafx.h et c'est tout. Donc, le fichier qui contient toutes les entêtes en parsé une fois et c'est tout.

Le préprocesseur

Hérité du langage C, le préprocesseur permet de changer le contenu du programme. #include ne fait qu'injecter le source dans le programme, et pour inclure plusieurs fois le même fichier dans un source, nous utilisons des barrières. Je m'explique... En réalité on ne met pas tous les headers dans les entêtes précompilées, on met juste les entêtes pour les fichiers .h définies par le système d'exploitation et les librairies tierces. Les structures que nous définissons doivent comporter une structure qui ressemble à cela :

```
// Book.h - this header contains the definition of an entity Book

#ifndef BOOK_HEADER
#define BOOK_HEADER
#include <string>

struct Book
{
   string name;
   int price;
   string comments;
};
#endif
```

Pour un compilateur qui ne supporte pas les entêtes précompilées, le fichier peut être inclus plusieurs fois, il ne sera parsé qu'une seule fois juste avec l'astuce d'un ifndef.

Le namespace std

Il est possible de définir des types dans des espaces de nom. Pour ce, il suffit de déclarer un bloc namespace <nom> {...} et tous les types seront dedans. C'est utile pour séparer les choses. La librairie standard du C++ (STL alias Standard Template Library) définit tous ses types dans le namespace std (standard). Pour inclure la résolution des noms dans un programme, il suffit d'écrire using namespace <nom> ;

Le type string

La STL définie le type string. C'est une classe template qui permet de gérer les chaînes de caractères. Avant de pouvoir utiliser string, il faut ajouter l'instruction #include <string>. string est définie dans le namespace std. Le type string est une classe donc il y a des méthodes pour manipuler les chaînes. On peut même utiliser l'opérateur + pour concaténer des chaînes. On peut utiliser la méthode c_str() pour obtenir le pointeur const sur la chaîne de caractères comme en C.

```
void Sample_String()
{
   string audrey = "je suis une coquine";
   string maggie = audrey + string(" a sa maman");
   if( !maggie.empty() )
   {
      cout << "taille: " << maggie.size() << endl;
   }
   for (auto c : maggie) // pour chaque charactère
   {
      cout << c;
   }
   cout << endl;

   const char * pChar = maggie.c_str();
   printf("%s\n", pChar);
}
```

La classe string est simple à utiliser.

Le type vector

C'est le type le plus connu de la STL avec string. vector<T> permet de stocker des éléments contigus. Si vous cherchez à stocker des éléments en mémoire, il faut utiliser le vector<T>

```
void Sample_Vector()
{
   vector<int> v = { 1, 2, 3, 4, 5, 6 };
   vector<string> girls;
   girls.push_back("Edith");
   girls.push_back("Lisa");
   girls.push_back("Audrey Maggie");

   string lacoquine = girls[2];
   cout << "la coquine c'est " << lacoquine << endl;

   for (auto & f : girls)
   {
      cout << f << endl;
   }
}
```

Les Itérateurs

La STL introduit la notion d'itérateurs qui permet, comme un pointeur, d'avoir un accès indirect à un objet. Dans le cas d'un itérateur, l'objet est un élément dans un container ou dans une chaîne string. Un itérateur permet de récupérer un élément et de passer la position suivante. Comme un pointeur, un itérateur peut être valide ou non. La fonction begin() permet d'obtenir la première position de l'itérateur et end() indique la dernière position invalide. Si le container est vide, begin() et end() auront la même valeur.

```
void Sample_Iterator()
{
   vector<string> girls = { "didou", "mini-mini", "méga teuteute" };
   vector<string>::const_iterator cit = begin(girls);
   cout << "my 12y baby is " << *cit << endl;
   ++cit;
   ++cit;
   cout << "super maggie is " << *cit << endl;
   ++cit;
   if (cit == end(girls))
   {
      cout << "iterator is end..." << endl;
   }

   // it est un vector<string>::iterator
   for (auto it = girls.begin(); it != girls.end(); ++it)
   {
      cout << *it << endl;
   }
}
```

Dans cet exemple, j'utilise un iterateur const et un itérateur normal. Voici les opérations possibles sur un itérateur de container :

Opération	Explication
*iter	Retourne une référence vers l'objet pointé par iter
Iter->mem	Accède à l'objet memoire
++iter	Avance d'une position pour aller sur le prochain objet
–iter	Recule d'une position pour avoir l'objet précédent
Iter1==iter2	Compare deux itérateurs
Iter1 !=iter2	Compare deux itérateurs

Il existe deux types d'itérateurs: const_iterator et iterator. Les operations begin() et end() peuvent s'écrire de deux façons :

```
   auto it1 = begin(girls);
   auto it2 = girls.begin();
   if (it1 == it2)
   {
      cout << "it1 == it2" << endl;
   }
```

Les opérations (on passe vite)

Un bloc en C++ commence par { et se termine par }. Il existe les opérations suivantes: if, switch, while, do while et for. Le C++ 11 introduit le for simplifié alias range-for :

```
void Sample_For()
{
   vector<string> girls = { "didou", "mini-mini", "maggie" };
   for (auto & girl : girls)
   {
```

```
        cout << girl << endl;
    }
}
```

D'autres opérations existent comme break, continue ou goto.

La gestion des exceptions

Les exceptions sont un bloc try..catch avec le type d'exception à catcher. Il est possible de lancer une exception via un throw. La STL définit la classe exception pour les problèmes principaux. Les exceptions sont définies dans std::except.

Les fonctions et le linker (édition de lien)

Pour définir une fonction, il faut un nom, un type de retour et éventuellement des arguments. Comme les programmes peuvent devenir complexes, on crée des fonctions dans des fichiers .cpp et on les utilise dans d'autres fichiers. C'est le principe de la compilation séparée. Pour pouvoir utiliser ce mécanisme, il faut que la fonction soit définie dans un point .h ou .cpp avec juste sa déclaration avec un ; (qu'on appelle le prototype). Cela permet au compilateur de savoir qu'un appel est fait à une fonction et c'est l'éditeur de lien (le link) qui va se charger de faire la résolution des trucs pour construire un exécutable.

Passage d'argument par références

Les habitués du C passent des pointeurs en paramètres alors que les habitués du C++ passent leur paramètre par références.

```
void Reset_Int_Ptr(int * i)
{
    *i = 0;
}

void Reset_Int_Ref(int & i)
{
    i = 0;
}

void Sample_Function()
{
    int a = 10;
    Reset_Int_Ptr(&a); // on passe l'adresse de a
    cout << a << endl;
    int b = 200;
    Reset_Int_Ref(b); // on passe b par reference
    cout << b << endl;
}
```

Gérer la ligne de commande

La ligne de commande du "main" est simple à gérer. Le premier argument donne le nombre d'éléments et le deuxième est un tableau de chaînes contenant les paramètres.

```
int main(int argc, char **argv) { ... }
```

Des fonctions avec des paramètres variables

Introduit avec le C++ 11, les fonctions à paramètres variables tirent parti des initializer_lists. Un type initializer_list est un type de a STL qui représente un tableau. Il est présent dans le header initializer_list.

```
void error_msg(initializer_list<string> il)
{
    for (auto beg = il.begin(); beg != il.end(); ++beg)
        cout << *beg << " ";
    cout << endl;
}

void Sample_Initializer_List()
{
    initializer_list<string> params = { "I am happy", "with a beer", "at 6PM" };
    error_msg(params);
}
```

Arguments de fonctions par défaut

Il est possible de donner une valeur par défaut aux paramètres d'une fonction. Ceci doit être spécifié dans la déclaration de fonction dans un fichier d'entête spécifique.

Fonctions inlines

Héritée du C, une fonction peut être annotée inline. Cela veut dire qu'à chaque appel dans le code source, la fonction sera éclatée en vrac avec son code. C'est pour gagner des pouillèmes en performance mais c'est très souvent utilisé. Ne méprisez pas.

Les macros du préprocesseur

Il existe 4 macros qui sont super utiles en debugging:

__FILE__ : nom du fichier source
__LINE__ : numéro de ligne
__DATE__ : date de compilation
__TIME__ : heure de compilation

Les pointeurs de fonctions

Un pointeur de fonction est un pointeur standard sauf qu'au lieu de pointer sur un objet, il pointe sur une fonction. Plus précisément, il pointe sur une fonction qui possède un type retour et des arguments d'un certain type. Le nom de la fonction n'entre pas en ligne de compte pour le pointeur.

```
bool CompareInteger(const int & a, const int & b)
{
    if (a > b)
        return true;
    else
        return false;
}

void Sample_FunctionPointer()
{
    // déclaration du pointeur de fonction
    bool(*pFn)(const int&, const int&);

    // association ptr
    pFn = &CompareInteger;

    if( (*pFn)(100, 99) == true )
    {
        cout << "superior !" << endl;
    }
}
```

La suite, c'est du lourd...

Dans l'article suivant, nous aborderons les classes qui sont l'essence de C++. Nous verrons les types simples, les constructeurs, les destructeurs, les méthodes virtuelles, l'héritage simple et multiple, les classes amies et peut-être les templates - ce sera peut-être un article à part – donc un beau programme en perspective !