Je pratique le C++

Partie 2/5

Nous vous proposons une série d'articles sur la pratique de C++ pour que vous puissiez tous vous y mettre. En C++, nous utilisons les classes pour définir nos propres types de données.

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

Cet article va vous donner une idée du support de l'abstraction et de la gestion des ressources en C++. Comment définir des nouveaux types définis par l'utilisateur mais aussi les propriétés basiques, les techniques d'implémentation et les possibilités du langage pour les classes concrètes, les classes abstraites et les hiérarchies de classes? Le langage supporte le style de programmation orientée objet et de programmation générique (avec les templates).

La fonctionnalité principale du langage C++ est la classe. Une classe est un type défini par l'utilisateur qui permet de représenter un concept dans le code d'un programme.

N'importe quel design d'un programme possède des concepts, des idées, des entités, etc. que nous essayons de traduire en classes de telle manière que la lisibilité, la maintenance et l'évolution du programme en soit améliorée. Un programme est un ensemble de classes définies par l'utilisateur pour faire une tâche bien précise. Les librairies sont des ensembles de classes mis à disposition.

On distingue deux catégories de développeurs : celui qui fait les classes et celui qui les utilise. L'approche est complètement différente. La plupart des techniques de programmation évoquent comment designer et implémenter des types de classes. Il existe 3 sortes de classes : les classes concrètes, les classes abstraites et les classes dans les hiérarchies de classes.

Nous allons nous attacher à l'importance de l'abstraction des données, ce qui permet de séparer l'implémentation d'un objet des opérations que cet objet peut effectuer. Les objets peuvent être copiés, déplacés ou détruits. Il est même possible de définir ses propres opérateurs. Les idées fondamentales derrières les classes sont l'abstraction des données et l'encapsulation. L'abstraction des données est une technique de programmation qui se base sur la séparation de l'interface et de l'implémentation. L'interface d'une classe présente les opérations qu'un utilisateur de la classe peut exécuter. L'implémentation contient les données membres de la classe, le corps des fonctions présentes dans l'interface et toutes les fonctions internes à la classe pour faire son job. L'encapsulation permet la séparation de l'interface de classe de son implémentation. Une classe cache son implémentation, et les utilisateurs de cette classe n'ont pas accès (des fois) à son implémentation. C'est le mécanisme des librairies dans lequel on ne possède que le point .h (header) de la classe, et l'implémentation est fournie sous forme de binaire (.dll).

Une classe possède un header ; c'est un fichier d'entête .h et un corps, c'est un fichier cpp. Par convention, il est possible que les headers soient déposés dans un répertoire INCLUDE et les fichiers CPP dans un dossier SRC. Vous allez me dire « ouaip mais en java ou en C#, on met tout dans la classe et puis c'est tout ! ». Oui c'est vrai mais en C++ on ne fait pas comme ça.

Examinons une classe de log toute simple au travers de son header. La classe est préfixée du nom de l'entreprise pour laquelle elle a été faite (Cerius) en 1995. Cette classe utilise un type CString. C'est un type qui provient d'une librarie très connue faite par Microsoft qui se nomme MFC (Microsoft Foundation Classes).

```
class CerLog
{
public:
   CerLog(const CString& path);
   ~CerLog();

   BOOL Log(ORB_REQUETE * pReq);

private:
   CString m_path;      // Répertoire du log
   static long m_stCompt;
};
```

Cette classe est dans un fichier CerLog.h qui est dans le répertoire INCLUDE. Que remarque t-on ? Elle possède un constructeur qui porte le nom de la classe. Ce constructeur sera appelé dès la création d'un objet. Le ctor prend une chaîne de caractères en paramètre. C'est obligatoire ; il n'y a pas de ctor vide. La première partie de la classe est dans un bloc public mais sur la fin on voit du private, ce qui implique que ce sont des données membres que l'on ne pourra pas utiliser. C'est réservé à la classe. Donc, dans cette classe, on a un ctor, un dtor (destructeur) et une méthode Log, c'est tout. Ouvrons le code de cette classe CerLog :

```
#include "cerlog.h"

long CerLog::m_stCompt = 0;

CerLog::CerLog(const CString& path) : m_path(path)
{
   SECURITY_ATTRIBUTES  sa;
…
   ::CreateDirectory(m_path,&sa);
}

CerLog::~CerLog()
{}

BOOL CerLog::Log(ORB_REQUETE * pReq)
{
   HANDLE hFic;
…

   bDone = ::WriteFile(hFic,(LPSTR) pReq,
         (DWORD)pReq->usLgRequete,
         &dwBytesWritten,
         NULL);
   ::CloseHandle(hFic);
return TRUE;
}
```

On remarque que le membre static est initialisé et on trouve le code du ctor, du dtor et la méthode Log. Rentrons dans les détails du fonctionnement d'une classe.

Définir une fonction membre

Il est possible de définir la signature dans le header et de mettre l'implémentation dans le fichier cpp. Mais pour certains membres, on peut les définir dans le header. Ainsi les propriétés simples ont leur place dans le header. Voici comment on aurait designer la classes CerLog2 pour une utilisation simple :

```
void Discover_Class()
{
    CerLog2 log("c:\\temp");
    log.Log("hello the logger");
}


class CerLog2
{
public:
    CerLog2(const string &path) : m_path(path) {}
    ~CerLog2() {}

    string GetPath() const
    {
        return m_path;
    }

    void Log(string message)
    {
        //...
    }

private:
    string m_path;
};
```

Et là vous me dites : « mais ça ressemble à du Java ou du C# ! ». En effet, si on met tout le code dans le header... Mais généralement on propose une solution avec deux personnages différents. Il y a celui qui construit et designe la classe et il y a celui (ou celle) qui l'utilise.

Introduction au this et const

This représente un pointeur sur l'objet à l'intérieur d'une classe.

```
string GetInternalPath() const { return this->m_path;  }
```

Le fait de préciser que la fonction membre est const indique que l'on ne peut pas modifier les valeurs des données membres. Le this devient un this const. Les objets qui sont const et les références ou pointeurs vers des objets const ne peuvent appeler que des fonctions membres const. Il faut noter que lorsqu'on écrit une fonction membre à l'extérieur du header, il faut préciser le nom de la classe avec :: et respecter les paramètres de la fonction.

Définir une fonction qui retourne l'objet « this »

Ajoutons une méthode Merge dans la classe MyLogger pour merger un logger.

```
class MyLogger
{
…
public:
    MyLogger& Merge(const MyLogger &logger);
…
};
```

Voici l'implémentation :

```
MyLogger& MyLogger::Merge(const MyLogger &logger)
{
    m_path = logger.m_path;
    return *this;
}
```

Le logger positionne le path et retourne un objet this dans sa totalité avec le * sur this. C'est une référence qui est retournée.

Définir des fonctions non membres d'une classe

Des fois, il est nécessaire de créer des fonctions auxiliaires (read,write, print) qui travaillent avec notre classe. Dans ce cas, il faut définir la fonction dans le même header que la classe.

Dans le header :

```
void LogAMessage(string message);
```

Dans le cpp :

```
void LogAMessage(string message)
{
    MyLogger logger("c:\\temp");
    logger.Log(message);
}
```

Le constructeur

Par défaut, le compilateur définit un ctor qui ne fait rien. Chaque classe définit comment les objets sont initialisés. Les classes contrôlent l'initialisation des objets en définissant une ou plusieurs fonctions membres qui portent le nom de la classe et ce sont des constructeurs (ctor). Le ctor initialise les données membres d'un objet de classe. Un constructeur est exécuté lorsque l'objet d'une classe est créé. Les constructeurs n'ont pas de type de retour et peuvent être un bloc vide. Une classe peut avoir plusieurs constructeurs qui diffèrent par leurs paramètres. Un constructeur ne peut pas être marqué const. La classe MyLogger peut avoir les ctor suivants :

```
MyLogger()
{
    m_path = "c:\\temp";
}

MyLogger(string path)
{
    m_path = path;
}
```

Les ctor et l'initialisation par liste

Il est possible de fournir une liste au ctor. C'est le C++ 11 qui permet cela. Dans le header :

```
MyLogger(string path, initializer_list<string> log_headers);
```

Dans le cpp:

```
MyLogger::MyLogger(string path, initializer_list<string> log_headers)
{
    m_path = path;
    for (auto it = log_headers.begin(); it != log_headers.end(); ++it)
    {
        Log(*it);
    }
}
```

Et voici comment utiliser ce ctor :

```
    MyLogger log4("c:\\temp", { "begin log", "1 juin 2015", "Application Totor" });
    log4.Log("the logger log4");
```

Contrôle d'accès et encapsulation

Lorsque nous définissons une interface pour notre classe, rien ne force l'utilisateur à respecter les appels dans le bon ordre ou le choix des méthodes. C'est la raison pour laquelle nous cachons l'implémentation dans des blocs private. Le contrôle d'accès garantit l'encapsulation. Les membres définis après public sont accessibles dans toutes les parties du programme. Les membres publics définissent l'interface de la classe. Les membres définis après private sont accessibles aux fonctions membres de la classe, mais ne sont pas accessibles au code qui utilise la classe. Les sections private encapsulent (cache) l'implémentation.

Class ou struct, il faut choisir

Class et struct ont le même sens si ce n'est que dans struct par défaut, tout est public, et que dans class, par défaut, tout est privé. Mais c'est la même chose.

Les fonctions ou classes friend (amies)

Reprenons la classe MyLogger et ajoutons-lui un membre private pour la sécurité (un exemple) :

```
class MyLogger
{
…
private:
    string m_path;
    SECURITY_ATTRIBUTES sa;
};
```

Et définissons la fonction LogAsAdministrator :

```
void LogAsAdministrator(string message)
{
    MyLogger logger("c:\\temp");
    // fake function :)
    logger.m_sa = ::CreateRestrictedToken(Windows::Administrator);
    logger.Log(message);
}
```

Cette fonction doit accéder au membre private m_sa qui est le jeton de sécurité. Problème, cette fonction n'est pas dans la classe. Donc, il faut la déclarer en friend (amie) et ainsi elle aura le droit d'accéder à tous les membres de la classe. Magique !

```
class MyLogger
{
    friend void LogAsAdministrator(string message);

…
```

La mécanique est la même pour les classes friend.
Je vais ajouter une classe qui fournit le privilège fictif administrator.
Dans le header de MyLogger :

```
    void LogAsAdministrator(const string & message);
private:
    CSecurityHelper m_sec;
};
```

Dans le cpp de MyLogger:

```
void MyLogger::LogAsAdministrator(const string & message)
{
    m_sec.m_sa = m_sa;
    m_sec.EnableAdministratorMode();
    Log(message);
}
```

La classe MyLogger au travers sa méthode LogAsAdministrator va renseigner un membre private de CSecurityHelper qui est m_sa en lui fournissant le sien pour demander une élévation de privilège.
Pour pouvoir accéder au membre privé, il faut que la classe MyLogger soit friend de CSecurityHelper dont voici le header :

```
#pragma once

class CSecurityHelper
{
    friend class MyLogger;

public:
    CSecurityHelper() {}
    ~CSecurityHelper() {}

    void EnableAdministratorMode();

private:
    SECURITY_ATTRIBUTES m_sa;
};
```

Voici le cpp :

```
#include "stdafx.h"
#include "MyLogger.h"
#include "SecurityHelper.h"


void CSecurityHelper::EnableAdministratorMode()
{
    // use m_sa
    // fake function :)
    m_sa.lpSecurityDescriptor = NULL; // ::CreateRestrictedToken(Windows::Administrator);

}
```

Pour que cela compile, il faut faire un #include de MyLogger.h pour que le compilateur connaisse la définition de la classe friend.

Les typedef dans les classes

Pour définir des types utilisateurs, on utilise parfois le typedef à l'intérieur d'une classe. Cela rend la classe plus lisible.
Reprenons la classe MyLogger qui possède des entêtes avant de logger, et que nous allons stocker dans un vector<string>.
Nous pouvons stocker le vector en private et déclarer les itérateurs comme typedef avec un nom plus simple...
Voici à quoi cela ressemble :

```
class MyLogger
{
...
public:
   typedef vector<string> HEADERS;
   typedef vector<string>::const_iterator CIT;

private:
   HEADERS m_headers;
```

Le typedef est là pour vous aider à rendre les types plus lisibles.

La résolution des noms

Le compilateur passe son temps à chercher les noms de fonctions qui matchent. De temps en temps dans le code on trouvera une ligne de code comme ::WriteFile() ; cela veut dire que le scope recherché est global et que cela ne se trouve pas dans la classe dans laquelle on utilise cette fonction.

Les membres static

Une classe peut avoir des membres static mais ils doivent être initialisés explicitement dans le fichier cpp. Exemple : dans le fichier header:

```
class NewLogger
{
public:
   static void Log(const string& message);
   static string m_path;
};
```

Dans le cpp :

```
string NewLogger::m_path = "c:\\temp";

void NewLogger::Log(const string& message)
{
   cout << message << endl;
}
```

Dans le programme qui l'utilise :

```
   NewLogger::Log("here is a static logger");
```

Il n'y a rien de compliqué.

La surcharge des opérateurs

Il est possible de surcharger tous les opérateurs de ++ en passant par -> en passant par [] ou ==.

```
bool operator==(const MyLogger &left, const MyLogger &right)
{
   return left.GetPath() == right.GetPath();
}
```

Introduction à la programmation orientée objet

Les idées clés dans la programmation orientée objet sont l'abstraction de données, l'héritage et le binding dynamic. Avec l'abstraction de données, on peut définir des classes qui ont une séparation entre l'interface et leur implémentation. Au travers de l'héritage, on peut définir des classes qui forment un modèle de relation avec des types similaires. Avec le binding dynamic, on peut utiliser des objets avec des types dont on ignore les différences par rapport à la classe de base. Prenons un exemple simple qui explique l'héritage, les classes abstraites et les fonctions virtuelles. Il y a la classe abstraite Animal et deux classes dérivées que sont Cat et Dog :

```
class Animal
{
public:
   Animal() {}
   virtual ~Animal() {}
public:
   virtual void Eat() = 0;
   virtual string Type() { return "Animal"; }
};

class Cat : public Animal
{
   virtual void Eat()
   {
      cout << "whyskas croquettes for Cat" << endl;
   }
   virtual string Type() { return "Cat"; }
};

class Dog : public Animal
{
public:
   virtual void Eat()
   {
      cout << "whyskas croquettes for Dog" << endl;
   }
   virtual string Type() { return "Dog"; }
};
```

J'ai volontairement tout mis dans le header pour faire plus concis. Voici ce qu'il faut retenir : la classe Animal est une classe abstraite car elle contient la méthode Eat() qui est virtuelle pure (=0) ; ça veut dire que toute classe qui hérite de Animal a l'obligation de redéfinir la méthode Eat. Le destructeur est annoté virtual : c'est obligatoire pour les dtor dans les classes mères et dérivées.

```
   Animal * ptrAnimal;
   Cat c1;
   Dog d1;
   // Pointe sur le Cat
   ptrAnimal = &c1;
   cout << ptrAnimal->Type() << endl;
   ptrAnimal->Eat();
   // Pointe sur le Dog
   ptrAnimal = &d1;
```

```
cout << ptrAnimal->Type() << endl;
ptrAnimal->Eat();
```

Ce petit bout de code montre comment faire un binding avec un pointeur sur la classe abstraite. On n'a pas le droit de déclarer un type Animal, car c'est une classe abstraite ; en revanche, on a le droit de s'en servir comme pointeur et de pointer sur des types enfants. On pointe sur un Cat puis sur un Dog, et les méthodes virtuelles sont appelées comme par magie.

Le concept clé : le polymorphisme en C++

L'idée clé derrière l'OOP est le polymorphisme. Ce mot est dérivé d'un mot grec qui veut dire plusieurs formes. On parle des types liés à l'héritage comme des types polymorphiques parce que nous pouvons utiliser plusieurs formes de ces types tout en ignorant les différences entre eux. Quand on appelle une fonction dans une classe de base au travers d'une référence ou d'un pointeur de la classe de base, on ne connait pas le type de l'objet sur lequel ce membre est appelé. L'objet peut être un objet de la classe de base ou un objet de la classe dérivée. Si la fonction est virtuelle, alors la décision de savoir quelle fonction va s'exécuter est repoussée à l'exécution. La version de la fonction virtuelle qui tourne est celle définie par le type d'objet avec lequel la référence est liée ou pour un pointeur sur le type d'objet pointé. D'un autre côté, les appels à des fonctions non virtuelles sont déterminés à la compilation.

Contrôle d'accès et héritage

Comme dans une classe pour cacher ses propres membres, chaque classe contrôle si ses membres sont accessibles à une classe dérivée. Une classes utilise protected pour les membres qu'elle veut accessibles à ses classes dérivées mais veut protéger depuis un accès général. L'accès protected se positionne entre le private et le public. Comme private, les membres protected sont inaccessibles aux utilisateurs de cette classe. Comme public, les membres protected sont accessibles aux membres et friends des classes dérivées de cette classe. Un membre d'une classe dérivée ou friend peut accéder aux membres protected de la classe de base seulement au travers d'un objet dérivé. La classe dérivée n'a aucun accès spécial sur les membres protected des objets de la classe de base.

Le ctor par copie

Le ctor par copie prend en paramètre une référence const sur l'objet.

```
class Foo {
public:
  Foo();          // constructeur par défaut
  Foo(const Foo&);  // constructeur de copie
  // ...
};
```

Si nous ne définissons pas de constructeur par copie, le compilateur en fournit un pour nous. Le type de chaque membre détermine comment le membre est copié : les membres de classe sont copiés par le ctor de copie de cette classe.

Initialisation par copie

Nous pouvons maintenant déterminer la différence entre l'initialisation directe et l'initialisation par copie.

```
string dots(10, '.');          // directe initialisation
string s(dots);                // directe initialisation
string s2 = dots;              // copie initialisation
string null_book = "9-999-99999-9"; // copie initialisation
string nines = string(100, '9');   // copie initialisation
```

Lorsque nous utilisons l'initialisation directe, nous demandons au compilateur de choisir un constructeur qui matche les paramètres que nous fournissons. Quand nous utilisons l'initialisation par copie, nous demandons au compilateur de copier l'opérande de droite dans l'objet à créer. L'initialisation par copie utilise le constructeur de copie.

L'opérateur de copie

Comme une classe contrôle comment les objets de cette classe sont initialisés, elle contrôle aussi comme les objets de cette classes sont assignés (=).

```
Book ref, primer ;
ref = primer ; // utilise l'opérateur de copie de Book
```

Comme avec le ctor par copie, le compilateur fournit un opérateur de copie si la classe ne définit pas le sien.
L'opérateur de copie prend en argument le même type que la classe :

```
class Foo {
public:
  Foo& operator=(const Foo&); // opérateur d'assignement (copie)
  // ...
};
```

Les opérateurs d'assignement doivent retourner une référence sur un type. Les classes qui ont besoin d'un destructeur ont besoin d'un ctor de copy et d'un opérateur d'assignement. C'est une règle en C++ qui permet de déterminer si on a besoin d'un opérateur de copie et d'assignation ; avant tout, on détermine si la classe a besoin d'un destructeur. Si on est sur qu'il faut un destructeur, alors c'est sur et certain qu'elle nécessite un ctor de copie et un opérateur d'assignement.

Conclusion

En C++, la classe est l'inséparable alliée de la programmation orientée objet. Cet article se concentre sur la manière d'expérimenter simplement les fonctionnalités offertes par une classe. Il y a encore beaucoup de choses à appréhender comme la sémantique de déplacement, la redéfinition des opérateurs, les fonctions virtuelles. Avec cet article, vous avez les bases pour faire votre propre construction.
Créez des classes, assemblez-les et n'oubliez pas : il y a deux casquettes, celui qui conçoit la classe et celui qu'il l'utilise. Il est bien plus simple d'être utilisateur que concepteur de classes. La maîtrise des principes de la programmation objet passe par de l'entraînement et de l'expérience. En C++, il n'y a jamais (ou presque) de classes deprecated. Donc une fois créée, la classe est là et pour longtemps. Prenez le framework de classes des MFC ; élaboré vers 1990, ce framework ne cesse de grossir et d'évoluer. Quand vous faites du C++, vous avez en tête que le code marche pour longtemps. C'est un exercice que l'on ne rencontre pas avec les langages dits modernes comme Java ou C#. En C++, on garde tout. La pyramide grossit mais on ne jette rien. Regardez vers le monde Linux : GTK, Kde, Xfce, toutes ces librairies évoluent depuis des années et sont toujours utilisées. Regardez QT, il ne s'est jamais aussi bien porté. Les ISV adorent QT car il permet de faire des GUI multiplateformes de folie. Allez, lancez-vous !