

# C++/CLI quoi de neuf ?

par Patrick OTTAVI MVP Visual C++ ([Site](#)) ([Blog](#))

Date de publication : 30/08/2007

Dernière mise à jour : 30/08/2007

Le C++/CLI est un langage à part parmi les langages .NET, il est le seul à permettre le mélange de code avec le langage dont il est issu le : C++.

Cette fonctionnalité est importante car elle permet la récupération de code C++ ou le rafraichissement de programmes développés avec les MFC ;

En effet depuis Visual 2005 il est très facile de mélanger du code C++/CLI avec un programme MFC, ce dernier bénéficiant alors de l'enrichissement de l'interface utilisateur avec la plateforme .net.

Un autre aspect, le C++/CLI est le langage de l'interopérabilité :

Avec le développement d'assembly il permet la récupération de code C++ encapsulé dans un wrapper de classe, permettant ainsi l'utilisation de code natif c++ dans les autres langages .net.

En dehors de ces aspects, les évolutions du langage C++ mises en #uvre dans le C++/CLI sont très intéressantes de part les apports syntaxiques ou les nouvelles fonctionnalités.

La plus visible étant le ramasse miette mémoire au niveau des objets managés. A travers cet article je vais décrire les différences et les apports par rapport au C++ standard.

Il est donc important pour pouvoir lire ces pages d'avoir une connaissance préalable du langage C++.

- I - Les types de données
  - A - Types de données utilisateur
    - I-A-1 - Type énuméré
    - I-A-2 - struct et value class
    - I-A-3 - Type référence
      - I-A-3.a - Les Tableaux
      - I-A-3.b - Interface
      - I-A-3.c - Delegates et Events
    - I-A-4 - Boxing et Unboxing
    - 1-A-5 - Type modificateurs et qualificateurs
    - 1-A-6 - Conversion de type
    - 1-A-7 - Les String
- II - Eléments spécifiques de syntaxe :
  - II-A - Adresse de référence et opérateur d'indirection
  - II-B - Référence et opérateur d'indirection en action
  - II-C - Construction des boucles
  - II-D - Les fonctions
    - II-D.1 - Passer un argument à une fonction
    - II-D.2 - Renvoyer une valeur depuis une fonction
    - II-D.3 - Retourner une référence
- III - Programmation objet
  - III-A - Généralités
  - III-B - Avantages et inconvénients
  - III-C - Les constructeurs
    - III-C.1 - Le constructeur de copie
    - III-C.2 - Le constructeur static
  - III-D - Initialisations des données membres statiques
  - III-E - Destructeurs
  - III-F - Finaliseur
  - III-G - Méthodes virtuelles
    - III-G.1 - Substitution implicite d'une fonction virtuelle
    - III-G.2 - Cacher la virtualité
    - III-G.3 - Substitution par nommage explicite d'une méthode virtuelle
    - III-G.4 - Méthode virtuelle pure
  - III-H - Méthode surchargée
    - III-H.1 - Surcharge d'opérateurs managés
    - III-H.2 - Surcharge d'opérateurs unaires
    - III-H.3 - Surcharge d'opérateurs binaires
  - III-I - Les propriétés membres
    - III-I.1 - Les Propriétés static
    - III-I.2 - Les propriétés tableau (array)
    - III-I.3 - Les propriétés indexées
    - III-I.4 - Les propriétés indexées par défaut
  - III-J - Gestion des transtypages entres classes
  - III-K - Les différents types de classes
    - III-K.1 - Les classes ref sealed
    - III-K.2 - Les classes ref Abstraites
    - III-K.3 - Classe Interface
    - III-K.4 - Les classes génériques
      - III-K.4.a - Définition d'une contrainte de type
    - III-K-5 - Les Templates
  - III-L - Les Exceptions
    - III-L.1 - Exception spécialisée
    - III-L.2 - La classe de base Exception du framework

III-L.2.a - Interception des exceptions utilisateurs

III-M - Delegates

III-N - Events

IV - Conclusion

V - Remerciements

## I - Les types de données

Tous les types de données sont initialisés à zéro, même les types classiques C++ puisqu'ils sont des alias d'objet du framework .net.

Type natif C++	Correspondance dans le Framework .Net
wchar_t	<b>System::Char</b> caractères 16 bits Unicode
unsigned char	<b>System::Byte</b>
signed char	<b>System::SByte</b>
double, long double	<b>System::Double</b>
float	<b>System::Single</b>
int, signed int, long, signed long	<b>System::Int32</b>
__int64, signed __int64	<b>System::Int64</b>
short, signed short	<b>System::Int16</b>
bool	<b>System::Boolean</b>

Tous les objets héritent de **System::Objet**

Méthodes communes :

<b>Object()</b>	création d'une nouvelle instance d'un objet
<b>Equals()</b>	comparaison de 2 instances pour voir si elles sont égales
<b>GetHashCode()</b>	renvoie un hascode
<b>GetType()</b>	le type de donnée d'un objet
<b>ReferenceEquals()</b>	vérifie si deux instances d'un objet sont les mêmes
<b>ToString()</b>	renvoie une chaîne qui représente l'objet.

## A - Types de données utilisateur

### I-A-1 - Type énuméré

**enum class** ou **enum struct** : nommage constant.

On peut utiliser **System::Enum** ou **enum** du c++

**Exemple :**

```
enum class SerialParity { None,Odd,Even};  
SerialParity Parity;  
Parity=SerialParity::Odd;
```

## I-A-2 - struct et value class

**value struct :**

**value class** : pareil sauf que par défaut les données de value struct sont publiques alors celles de value class sont privées.

**value struct** et **value class** ne sont pas managées, elles ne bénéficient pas du ramasse miette mémoire.

**value class** et **value struct** héritent de la classe .net **System::ValueType**.

Elle permet que toutes les valeurs soient déposables sur la pile.

Elles peuvent hériter uniquement d'interface.

Tenter d'hériter de value struct ou value class provoquera une erreur à la compilation.

Ces deux types sont à réserver pour des types de données simples (pod).

La surcharge de l'opérateur d'affectation n'est pas autorisée, la copie des données se fait donc membre à membre.

## I-A-3 - Type référence

Les types de données référence sont des types que le programmeur développe et sont accessibles par des handles , ils sont stockés dans le tas managé.

Tous les types référence en C++/CLI sont garbage collector.

C++/CLI possède quatre types référence utilisateur définissables : **arrays, classes, interface, delegates**.

Ces quatre types ont une chose en commun : pour créer une instance ils requièrent l'opérateur **gcnew**.

**gcnew** utilise le symbol **^** qui est le pendant de **\*** avec **new** du C++

### I-A-3.a - Les Tableaux

Les arrays sont similaires aux tableaux en C++, ils utilisent la CRT tas memory.

Ils récupèrent la place perdue (garbage collector) et gèrent une dimension.

Ils peuvent gérer tous les types hérités de **System::Object** celle-ci étant la classe de base des objets du framework.

La déclaration d'un array nécessite un handle qui sera alloué dans le tas managé par **gcnew**.

### Syntaxe générale:

```
array<datatype>^ arrayVarName ;
```

```
//exemple de creation d'un array avec appel du constructeur :  
using namespace stdcli ::language ;  
array<double>^ ArrayDouble= gcnew array<double>(5); //5 double  
array<String^>^ Arraystrings= gcnew array<String^>(10);
```

Il est possible de gérer des données non managées dans un array à condition que le type de données soit de type pointeur.

```
class NoManage{} ;  
array< NoManage *>^ pNoManage = gcnew array< NoManage *>(10) ;  
for(int i=0;i<pNoManage ->Length;i++)  
    pNoManage [i]=new NoManage();  
for(int i=0;i<pNoManage->Length;i++)  
    delete pNoManage[i];
```

Il est possible d'initialiser un tableau en même temps que la déclaration :

```
array<String^>^ ArrayString= gcnew array<String^>{"1","2","3","4"};
```

Les tableaux multi-dimension se déclarent comme pour les tableaux de template.

Il suffit de rajouter le rang derrière le type de données, le rang est compris entre 1 et 32 éléments, une erreur sera générée pour les autres valeurs.

Le rang ne peut être une variable.

```
array<int,1>^ TwoInts= gcnew array<int>(2); //2 entiers  
array<int,2>^ TwoIntsx3= gcnew array<int,2>(2,3); // 2 X 3  
for(int n=0;n<TwoIntsx3->GetLength(0);n++)  
{  
    for(int i=0;i<TwoIntsx3->GetLength(1);i++)  
        TwoIntsx3[n,i]=((n*5)+i);  
}
```

Les tableaux ainsi déclarés ont une taille uniforme sur les dimensions, un tableau à deux dimensions sera rectangulaire.

Il est possible d'avoir un tableau qui n'a pas la même longueur dans les autres dimensions pour cela on utilisera la syntaxe connue en C++ :

### Syntaxe générale:

```
array< array<datatype>^ >^
```

```
array<array<int>^ >^ varMulti= gcnew array<array<int>^ >(2);  
for(int i=0;i<varMulti->Length;i++)  
varMulti[i] = gcnew array <int>(10+(i*3)); // 2 tableaux un de 10 et un de 13 elements
```

L'accès à une variable est réalisée en utilisant l'opérateur `[]` comme en C++

La nouveauté c'est lorsque le tableau est à plusieurs dimensions on écrira :

```
variable[index1,index2,...]
```

### I-A-3.b - Interface

Une interface est une collection de méthodes et de propriétés sans définitions.

L'interface n'a pas d'implémentation pour ses méthodes et ses propriétés.

### I-A-3.c - Delegates et Events

**Delegate** est un type de référence qui fonctionne comme un pointeur de fonction, qui peut être lié à une instance ou une méthode statique d'une classe C++/CLI **Delegate** peut être utilisé quand une méthode à besoin d'un appel dynamique et est utilisée comme callback de fonction pour intercepter les événements avec les applications .net.

Un **event** est une implémentation spécialisée d'un **delegate**.

Un **event** permet à une classe de déclencher l'exécution d'une méthode trouvée dans une autre classe, sans rien connaître sur cette classe.

Ces éléments seront développés dans les pages qui suivent.

### I-A-4 - Boxing et Unboxing

Technique permettant de convertir des types valeurs en types références.

Unboxing étant l'inverse.

La pile est la forme de défaut de stockage pour les types de valeur du framework.

Dans cette forme un type valeur ne peut accéder à ses méthodes comme **ToString()**, parce que le type de valeur doit être au format d'objet (référence).

Pour remédier à ce problème, le type de valeur est implicitement (automatiquement) boxed quand la méthode **ToString** est appelée.

Avec .net 2.0 tous les types valeurs sont implicitement boxed .

**Exemple :**

```
value class Point
{
public
    int x,y ;
    Point(int x,int y) :x(x),y(y){}
} ;
Point p1(1,2) ;
Object ^o =p1 ; // la conversion est implicite.
Point ^hp=p1;
Object ^ol = 1024; // boxing implicite
```

L'objet créé est une copie du type valeur, donc toutes les modifications faites dans l'objet boxed ne seront pas répercutées dans l'objet d'origine.

Unboxing un type référence en un type valeur requière un cast explicite

```
Point p2=(Point)o ;
Point p1=(Point)hp ;
```

## 1-A-5 - Type modificateurs et qualificateurs

Trois modificateurs et un qualificateur sont disponibles en C++/CLI, ils donnent un plus d'information sur la variable définie qui le précède.

### **auto :**

Permet de spécifier que la portée de la variable est le block où elle est définie.

Ce modificateur est optionnel.

```
auto int normalinteger ;
```

### **const :**

Identique au C++, une constante ne peut être modifiée même par pointeur interposé.

```
const Int32 intergerconstant=32 ;
```

Le C++/CLI ne supporte pas const sur les méthodes membres des types de données managés.

### **Exemple :**

```
bool GetFlag() const {return true ;}
```

N'est pas permis avec une classe déclarée value ou ref (value struct ou ref class) Ce n'est pas supporté sur une interface#

**extern :**

Identique au C++

**static :**

identique au C++

## 1-A-6 - Conversion de type

Quand la variable de réception est plus petite que la variable d'origine sur une opération d'affectation il y a une conversion de type pouvant conduire à une perte d'information.

**Exemple :**

```
UInt16 n=45000 ;  
Byte c=n ; // c vaut 175 et pas 45000.
```

Le compilateur vous signalera cet état par un warning, Pour enlever le warning il faudra utiliser un cast explicite .  
On utilisera :

```
safe_cast< type-de-donnée-pour-conversion >(expression).
```

Ou le bon vieux cast du C:

(type-de-donnée-pour-conversion) expression.

```
char b= safe_cast<char>(a);  
// ou  
char b=(char )a;
```

**Littéral:**

Les nombres numériques peuvent être exprimés comme en c++, En octal, entier, hexadécimal, décimal, exponentiel.

La nouveauté c'est qu'un littéral numérique peut être considéré comme un objet.

Le nombre devant être entouré par des parenthèses, ce qui permettra d'utiliser la méthode **ToString()**.

```
Console ::WriteLine((12356).ToString()) ;  
Console ::WriteLine((0xFF).ToString());
```

**Littéral booleen :**

Il y a deux littéraux pour le type bool : **true** et **false**.

Les deux sont aussi considérés comme des objets#

```
Console ::WriteLine(true.ToString());  
Console ::WriteLine(false.ToString());
```

## 1-A-7 - Les String

Les chaînes de caractères sont entourées de double quotes comme en C/C++, une chaîne créée avec le préfixe **L** sera traitée comme de l'Unicode.

```
String ^ s1= " \x61 " ; //a  
String^ s2="\x612"; // ne vaut pas a2 mais la valeur hexa de la séquence escape Unicode 612
```

## II - Eléments spécifiques de syntaxe :

### II-A - Adresse de référence et opérateur d'indirection

Opérateur	Action
&	adresse de
%	référence
*	indirection

L'opérateur adresse de (&) part sa nature d'être un manipulateur de pointeurs, doit être, et est une opération peu sûre (unsafe).

L'opérateur référence a été introduit par nécessité dans le C++/CLI, c'est une conséquence d'un manque syntaxique pour un opérateur sûr pour référencer un handle.

Cette opérateur fournit donc une nette différence syntaxique pour les objets managés et le code non protégé qui utilise l'opérateur adresse de (&).

```
int intvar=10 ;
int %intref =intvar ;
Console ::WriteLine(intref) ;
Intref=20 ;
Console ::WriteLine(intref) ;
```

### II-B - Référence et opérateur d'indirection en action

```
ref class Indirectionclass
{
public :
int n ;
Indirectionclass(int x){n=x;}
} ;

Indirectionclass LocalObj(10) ;
Indirectionclass ^p ;

p =%LocalObj ; // place une référence de LocalObj dans le handle p
Console ::WriteLine(p->n) ; // renvoie 10

int %i=LocalObj.n ; // LocalObj.n a une reference, attention ne fonctionne pas sur une propriété (
property int n )

i=50;
Console ::WriteLine(p->n) ; // renvoie 50

int ^y= gnew int(10); // attention c'est bien un int initialisé a 10 et pas un tableau de 10 int..
Console ::WriteLine(y) ;// 10
*y=500 ; // nouvelle valeur par indirection
Console ::WriteLine(*y) ;
```

Ce dernier code ne sera pas autorisé si on utilise l'option /clr :safe option.

## II-C - Construction des boucles

Quatre types de boucles sont disponibles: **while,do while,for,for each**.

**for each** habituellement réservé au Visual basic et au C#, permet l'itération à travers une collection dérivée de l'interface **IEnumerable**.

Les tableaux en sont un exemple :

```
array <int>^ arraynumbers= gnew array<int>{1,2,3} ;
for each (int i in arraynumbers)
{
    Console ::WriteLine(i) ;
}
```

Pendant l'itération il n'est pas possible de rajouter ou supprimer une occurrence de la collection.

Essayer lévera une exception..

## II-D - Les fonctions

### II-D.1 - Passer un argument à une fonction

Il y a deux manières de passer un argument à une fonction par valeur et par référence.

Par rapport au C++ nous avons une différence avec l'ajout d'un nouvel opérateur %

**Rappel** : par valeur, une copie de la variable est passée à la fonction, la modification de celle-ci dans la fonction n'affecte pas la variable d'origine.

Pour que la variable d'origine soit modifiée nous avons deux manières de procéder.

La première est de passer un handle par valeur, mais le problème c'est que la valeur d'origine ne sera toujours pas modifiée, il faudra adopter une syntaxe plus compliquée parce qu'il faudra déréférencer le handle :

```
void function(int ^v)
{
    *v= *v + 100 ;
}
int ^nvar=5 ;
function(nvar) ; // nvar vaut 105.
```

La seconde approche est le passage des arguments par référence.

**Rappel** : les variables passées par référence ne sont pas copiées, la fonction accède à un alias de l'argument, au final on peut dire que la fonction manipule l'argument directement.

```
void function(int %v)
{
    v= v + 100 ;
}
int nvar=5 ;
function(nvar) ; // nvar vaut 105.
```

## II-D.2 - Renvoyer une valeur depuis une fonction

On retrouve les mêmes pièges qu'en C et C++ sur le retour de l'adresse d'une variable déclarée localement dans une fonction.

Exemple :

Il faut faire attention lorsque l'on retourne un handle depuis une fonction.

Il ne faudra jamais faire ceci :

```
ref class MyClass{} ;

MyClass ^ function()
{
    MyClass var ;
    return %var ;
}
```

La variable var est déclarée localement à la fonction et n'existe plus en sortant de la fonction#

Par contre on pourra par exemple retourner un handle qui est passé en argument ou alors un handle qui est créé avec l'opérateur **gnew** dans la fonction.

```
ref class MyClass{} ;

MyClass ^ function(MyClass ^ var)
{
    return var ; // ok
}
```

Ou :

```
MyClass ^ function()
{
    MyClass ^ var= gnew MyClass();
    return var ; // ok
}
```

## II-D.3 - Retourner une référence

Même remarque que pour le handle : il ne faut jamais retourner une référence d'une variable déclarée localement dans une fonction.

```
ref class MyClass{} ;
```

```
MyClass % function()  
{  
  MyClass var ;  
  return var ;  
}
```

La variable " var " est déclarée localement à la fonction et n'existe plus en sortant de la fonction#

A la place on pourra retourner un argument passé par référence à la fonction, ou un pointeur sur une référence créée dans la fonction avec l'opérateur **gnew**.

```
ref class MyClass{} ;  
  
MyClass % function(MyClass % var)  
{  
  return var ; // ok  
}
```

Ou :

```
MyClass % function()  
{  
    MyClass ^ var= gnew MyClass();  
  return *var ; // ok  
}
```

## III - Programmation objet

### III-A - Généralités

Pour les programmeurs issus du C++, le mot clef " ref " n'est pas passé inaperçu, C'est le plus grand et le plus important changement par rapport à un programme C++.

L'utilisation du mot " **ref** " dit au compilateur que la classe doit être un objet référence sur le tas managé.

Le C++/CLI pour être en accord avec le C++, place par défaut les objets sur le tas CRT et non sur le tas managé.

Si on veut que la classe soit managée il faudra placer " ref " devant la classe.

### III-B - Avantages et inconvénients

**En contrepartie une classe managée apporte les avantages suivants :**

Ramasse miette mémoire (garbage collector).

Héritage depuis n'importe quelle classe de base du framework .net qui n'est pas scellée ou stoppée (sealed), si aucune classe de base n'est spécifiée l'héritage sera automatiquement fait sur la classe **System::Object**.

Possibilité d'utiliser les ref class avec les collections et tableaux du framework.

Héritage multiple depuis des interfaces managées.

Habilité à contenir des propriétés.

Habilité à contenir des pointeurs vers des classes non managées.

**Les moins :**

Héritage simple.

L'héritage depuis une classe non managée n'est pas permis.

Pour les **ref class** :

Ne peut être un parent d'un type non managé.

Ne peut contenir une surcharge de l'opérateur **gcnew** ou **delete** .

Doit utiliser un héritage public

Ne peut être utilisée avec l'opérateur **sizeof** ou **offsetof** .

Les manipulations arithmétiques sur les handles d'une classe ref ne sont pas permises.

Pas de surcharge d'une méthode avec des arguments par défaut.

### III-C - Les constructeurs

Un constructeur est appelé quand une nouvelle instance d'une ref classe est créée, celle-ci est allouée sur le tas managé qui est maintenu par la CLR.

Une différence de taille par rapport aux classes non managées, les variables sont initialisées à zéro avant l'appel effectif du constructeur.

Les initialisations spécifiques seront donc à faire dans le constructeur.

#### III-C.1 - Le constructeur de copie

Le constructeur de copie existe aussi en C++/CLI, la seule différence est l'utilisation de l'opérateur " %" à la place de " & " .

#### III-C.2 - Le constructeur static

C'est une nouveauté par rapport au C++, il est possible de définir un constructeur statique dans une ref class pour initialiser les variables statiques de la classe.

Si des initialisations de variables statiques ont été déjà spécifiées dans la classe leur contenu sera effacé.

### III-D - Initialisations des données membres statiques

Une ref classe accepte les fonctions membres statiques comme en c++, Ainsi que les données membres static.

Celles-ci devront être initialisées dans la classe.

```
ref class sampleclass
{
    static int nvar=3 ;
};
```

### III-E - Destructeurs

Les destructeurs ont deux objectifs, la libération des ressources allouées par l'opérateur new ou gnew , Et l'éventuelle libération de ressources systèmes.

En ce qui concerne les données allouées par gnew, elles peuvent être libérées par l'appel à l'opérateur delete comme en C++.

Il ne faut pas oublier que le job du CLR est à même de détecter lorsqu'un objet n'est plus utilisé et de le libérer, de ce fait le meilleur choix est de le laisser faire son boulot et d'omettre les delete pour les allocations d'objets managés.

### III-F - Finaliseur

Le C++/Cli dispose d'un autre destructeur appelé finaliseur, Celui-ci est appelé par la CLR quand il détecte que l'objet n'est plus utilisé.

La CLR avant son appel vérifie si l'opérateur delete n'a pas déjà été appelé, si c'est le cas il ne perd pas de temps à appeler le finaliseur #

Le finaliseur a la même syntaxe que le destructeur classique si ce n'est qu'il utilise le symbole " ! " à la place du " ~ ", son accès est protégé.

```
protected :  
    !MyClass() {}
```

En résumé, dans le destructeur classique on pourra :

Libérer toutes les ressources managées et non managées et la mémoire.

Pour le finaliseur :

On libérera uniquement les ressources non managées et la mémoire.

Dans la pratique on écrira une méthode pour libérer la mémoire sur les ressources non managées, celle-ci devra être appelée dans les deux destructeurs#

### III-G - Méthodes virtuelles

Il y a deux méthodes pour substituer/redéfinir (overriding) une fonction virtuelle : une implicite ou une explicite par nommage.

On peut cacher la fonction virtuelle et en démarrer une nouvelle, ou tout simplement stopper la séquence virtuelle de partout.

#### III-G.1 - Substitution implicite d'une fonction virtuelle

Pour une surcharge d'une méthode celle-ci doit disposer du même nom que celle de la classe de base qui inclut le préfixe **virtual**.

Le nom de la méthode et les arguments doivent être identiques.

Le type de retour peut ne pas être identique mais il doit dériver du même type que celui de la classe de base.

La nouveauté, il faudra aussi indiquer le mot clef override après les paramètres.

**Exemple :**

```
virtual void function() override
```

Le compilateur se chargera assez bien de vous le rappeler en cas d'omission.

```
ref class Tools
{
public:
    virtual void InitTools()
    {
        Console::WriteLine("Tools:InitTools");
    }
};
ref class Brush : public Tools
{
public:
    virtual void InitTools() override
    {
        Console::WriteLine("Brush:InitTools");
    }
};
int main(array<System::String ^> ^args)
{
    Tools ^p1= gnew Tools;
    p1->InitTools(); //Tools:InitTools
    Tools ^p2= gnew Brush;
    p2->InitTools(); // Brush:InitTools
    return 0;
}
```

### III-G.2 - Cacher la virtualité

Lorsque la classe parent définit une fonction virtuelle, ce n'est généralement pas sans raison.

Si l'on souhaite annuler la propagation de la virtualité d'une méthode dans les classes filles, on ajoutera le mot clef " **new** " après les arguments.

**Exemple :****virtual void function() new**

```
ref class Brush : public Tools
{
public:
    virtual void InitTools() new
    {
        Console::WriteLine("Brush:InitTools");
    }
};
int main(array<System::String ^> ^args)
{
    Tools ^p1= gnew Tools;
    p1->InitTools(); //Tools:InitTools
    Tools ^p2= gnew Brush;
    p2->InitTools(); // Tools:InitTools
    return 0;
}
```

}

### III-G.3 - Substitution par nommage explicite d'une méthode virtuelle

La substitution explicite nommée permet d'assigner un nom de méthode différent de la fonction virtuelle d'origine.

La nouvelle fonction sera déclarée virtuelle et l'on lui assignera le nom de la méthode virtuelle qu'elle doit substituer.

```
ref class Tools
{
public:
    virtual void InitTools()
    {
        Console::WriteLine("Tools:InitTools");
    }
};
ref class Brush : public Tools
{
public:
    virtual void CreateBrush() = Tools::InitTools
    {
        Tools::InitTools();
        Console::WriteLine("Brush:CreateBrush");
    }
};
int main(array<System::String ^> ^args)
{
    Tools ^p1= gcnew Tools;
    p1->InitTools(); //Tools:InitTools
    Tools ^p2= gcnew Brush;
    p2->InitTools();//Brush:CreateBrush
    return 0;
}
```

Un peu plus compliqué, il est possible d'exprimer deux surcharges différentes pour une seule méthode virtuelle.

```
ref class Tools
{
public:
    virtual void InitTools()
    {
        Console::WriteLine("Tools:InitTools");
    }
};

ref class ContextBar: public Tools
{
public:
    virtual void InitTools() new
    {
        Console::WriteLine("ContextBar:InitTools");
    }
};

ref class ToolBar : public ContextBar
{
public:
    virtual void InitTools() override = Tools::InitTools
    {
        Console::WriteLine("ToolBar:InitTools");
    }
};
```

```
int main(array<System::String ^> ^args)
{
    Tools ^p1= gnew ContextBar;
    p1->InitTools();
    ContextBar ^p2= gnew ContextBar;
    p2->InitTools(); //ContextBar:InitTools

    Tools ^p3= gnew ContextBar;
    p3->InitTools(); //Tools:InitTools

    ToolBar ^p4= gnew ToolBar;
    p4->InitTools(); //ToolBar:InitTools
    Tools ^p5=gnew ToolBar;
    p5->InitTools(); //ToolBar:InitTools

    ContextBar ^p6=gnew ToolBar;
    p6->InitTools(); //ToolBar:InitTools
    return 0;
}
```

On peut obtenir le même résultat en spécifiant la liste des surcharges, La seule contrainte étant de spécifier un autre nom pour la fonction de surcharge.

```
ref class Tools
{
public:
    virtual void InitTools()
    {
        Console::WriteLine("Tools:InitTools");
    }
};
ref class ContextBar: public Tools
{
public:
    virtual void InitTools() new
    {
        Console::WriteLine("ContextBar:InitTools");
    }
};
ref class ToolBar : public ContextBar
{
public:
    virtual void InitToolBar() = Tools::InitTools,ContextBar::InitTools
    {
        Console::WriteLine("ToolBar:InitTools");
    }
};
```

### III-G.4 - Méthode virtuelle pure

Comme en C++ il est possible de définir une méthode virtuelle pure pour forcer la définition de cette méthode dans les classes filles.

Une méthode virtuelle pure dans une ref classe class interdit d'instancier cette classe.

La classe doit donc être héritée pour être utilisée#

Dernier point une méthode virtuelle pure ne peut être cachée avec l'opérateur "new".

**Rappel syntaxique :**

```
virtual void PureFunction() = 0 ;
```

### III-H - Méthode surchargée

Pour les habitués du C++ les méthodes surchargées n'ont plus de secrets, et l'utilisation des arguments par défaut est un moyen simple d'y parvenir.

Malheureusement en C++/CLI les ref classes ne supportent pas les arguments par défaut !

La seule solution qui nous reste est de faire différentes méthodes surchargées du même nom .

Pas terrible#

#### III-H.1 - Surcharge d'opérateurs managés

La ref class du C++/CLI supporte la surcharge des opérateurs comme en C++, Mais avec une syntaxe un peu différente.

Les opérateurs surchargés managés doivent être déclarés static, la conséquence étant qu'il faudra pour :

Les opérateurs binaires passer les deux arguments dans la fonction.

Pour les unaires passer le paramètre de gauche à la fonction.

Cette syntaxe est particulière si on désire garder la compatibilité en développement multi-langages.

Sinon la syntaxe traditionnelle reste disponible.

A travers cette possibilité on voit bien que C++/CLI se positionne en tant que langage d'interopérabilité.

**Exemple d'opérateur surchargé:**

```
void operator *=(const MyClass ^ righth); // syntaxe classique.  
static void operator *=(const MyClass ^ lefth, const MyClass ^ righth); // syntaxe managée
```

#### III-H.2 - Surcharge d'opérateurs unaires

Les opérateurs unaires sont des opérateurs qui ne prennent qu'un argument : !,~,++,--,\*,%,&.

On pourra les écrire selon deux formes, une traditionnelle avec l'argument constant, l'autre dite mutable.

**Attention:** les opérateurs \*,%,& sont unaires et binaires suivant le contexte.

'\*' correspond à l'opérateur de multiplication (binaire) mais aussi à l'opérateur d'indirection (unaire).

De même que & désigne trois choses différentes, il est utilisé pour la spécification de référence, il désigne aussi l'opérateur logique ET bit à bit, et il représente l'opérateur "adresse de" l'ambiguïté sera levée suivant le contexte d'utilisation.

```
static MyClass ^operator -(const MyClass ^lefth) // opérateur changement de signe en -
{
    MyClass ^ ret=gcnew MyClass();
    ret->i--(lefth->i);
    return ret;
}
```

**La forme mutable:**

```
static MyClass ^operator -( MyClass ^lefth)
{
    lefth->i--(lefth->i);
    return lefth;
}
```

La forme " const " est toutefois préférable.

### III-H.3 - Surcharge d'opérateurs binaires

Les opérateurs binaires sont des opérateurs qui prennent deux arguments.

```
ref class MyClass
{
public:
    MyClass(int x){m_nx=x;}
    MyClass(){}
    static bool operator ==(const MyClass ^lefth,const MyClass ^righth)
    {
        return lefth->m_nx==righth->m_nx;
    }
    static bool operator ==(const MyClass ^lefth,int n)
    {
        return lefth->m_nx==n;
    }
    static bool operator !=(const MyClass ^lefth,const MyClass ^righth)
    {
        return lefth->m_nx!=righth->m_nx;
    }

    static MyClass ^ operator --(const MyClass ^lefth,const MyClass ^righth)
    {
        MyClass ^ret=gcnew MyClass;
        ret->m_nx=(lefth->m_nx-righth->m_nx);
        return ret;
    }

    static void operator --(MyClass ^lefth,int n)
    {
        lefth->m_nx-=n;
    }
    void operator *=(const int n) // syntaxe classique.
```

```

{
    m_nx*=n;
}
int m_nx;
};
int main(array<System::String ^> ^args)
{
    MyClass ^obj= gcnew MyClass(10);
    MyClass ^obj2= gcnew MyClass;
    Console ::WriteLine("obj==obj2 "+(obj==obj2).ToString());
    Console ::WriteLine("obj==10 "+(obj==10).ToString());
    obj2->m_nx=10;
    Console ::WriteLine("obj!=obj2 "+(obj!=obj2).ToString());
    obj2->m_nx=5;
    MyClass ^ obj3=obj2--obj;
    obj-=8;
    obj*=2;
    Console ::WriteLine("obj "+(obj->m_nx).ToString());
    Console ::WriteLine("obj2 "+(obj2->m_nx).ToString());
    Console ::WriteLine("obj3 "+(obj3->m_nx).ToString());

    return 0;
}

```

### III-I - Les propriétés membres

Le but des propriétés est de permettre une encapsulation de variables membres en les cachant à l'utilisateur tout en permettant leurs accès contrôlés par des méthodes d'accesseurs et modificateurs (getter and setter).

Evitant du coup de définir des fonctions spécifiques pour permettre leur accès comme on le ferait en C++ classique.

L'accès aux variables se faisant par leurs noms, l'utilisation en est simplifiée.

Pour déclarer une simple variable propriété, on placera le mot clef "**property**" devant sa définition :

#### property typevar PropertyVar

Une variable "**property**" va nous permettre de contrôler l'accès à sa variable en donnant des accès lecture seule ou écriture seule ou les deux :

```

property type PropertyVarName
{
    type get(){}
    void set(type value){}
}

```

#### Exemple:

```

ref class RefClass
{
public:
    RefClass(int x){m_x=x;}
    property int Intx
    {
        int get(){return m_x;}
        void set(int value){m_x=value;}
    }
}

```

```
}  
  
void Show()  
{  
    Console::WriteLine(m_x);  
}  
private:  
    int m_x; // variable cachée pour l'utilisateur  
};  
  
RefClass r(8);  
r.Intx=10;  
Console::WriteLine(r.Intx);  
r.Show();
```

Dans cet exemple aucun traitement ou contrôle n'a été réalisé, on s'est contenté d'affecter ou de récupérer la valeur de la variable interne dans ce cas précis la simple déclaration de la variable en tant que propriété suffit :

```
ref class RefClass  
{  
public:  
    RefClass(int x){ Intx =x;}  
    property int Intx;  
  
    void Show()  
    {  
        Console::WriteLine(Intx);  
    }  
};  
  
RefClass ^p= gcnew RefClass(10);  
p->Show();  
p->Intx=20;  
p->Show();
```

### III-I.1 - Les Propriétés static

Il est tout à fait possible de définir une propriété statique, il suffira de rajouter le mot clef **static** derrière "**property**".

La syntaxe est la suivante :

```
property static TypeVar PropertyName  
{  
    TypeVar TypeVar get{}  
    void set(TypeVar value){}  
}
```

### III-I.2 - Les propriétés tableau (array)

Le C++/CLI fournit une syntaxe simple pour un tableau de propriétés.

**Exemple :**

```
property array < int >^ nArray  
{  
    array < int >^ get{}
```

```
void set(array < int >^ value)
}
```

### Exemple:

```
ref class MyClass
{
public:
  MyClass(int nSize)
  {
    m_nIntArray= gcnew array< int >(nSize);
  }
  property array < int >^ nIntArray
  {
    array < int >^ get()
    {
      return m_nIntArray;
    }
    void set(array < int >^ value
    {
      m_nIntArray=value;
    }
  }

private:
  array < int >^ m_nIntArray; // variable cachée pour l'utilisateur
};
int main(array<System::String ^> ^args)
{
  MyClass Obj(10);
  for(int i=0;i< Obj.nIntArray->Length;i++)  Obj.nIntArray[i]=(i*2);
  return 0;
}
```

## III-I.3 - Les propriétés indexées

Les propriétés indexées permettent de gérer l'indexation sur une variable propriété.

Qu'elle est la différence avec une propriété tableau ?

Alors que la propriété tableau fournit la variable tableau interne, les propriétés indexées permettent de gérer l'accès à l'élément, la variable interne ne sera donc pas forcément un tableau #, ou permettra tout simplement de s'assurer que l'index demandé pour un tableau soit correct.

### Syntaxe :

```
Property TypeVar PropertyName [indexType,.. .,indexTypex]
{
  TypeVar get(indexType,.. .,indexTypex){};
  void set(indexType,.. .,indexTypex,TypeVar value){};
}
```

Exemple avec le contrôle des bornes d'un tableau:

```
ref class MyClass
```

```

{
public:
  MyClass(int nSize)
  {
    m_nIntArray= gnew array< int >(nSize);
  }
  property int nIntArray [ int ]
  {
    int get(int n)
    {
      if(n<0) n=0;
      else
      if(n>=m_nIntArray->Length) n=m_nIntArray->Length-1;

      return m_nIntArray[n];
    }
    void set(int nIndex,int nvalue)
    {
      if(nIndex<0) nIndex=0;
      else
      if(nIndex>=m_nIntArray->Length) nIndex=m_nIntArray->Length-1;
      m_nIntArray[nIndex]=nvalue;
    }
  }
  int GetSizeArray(){return m_nIntArray->Length;}
private:
  array < int > ^ m_nIntArray; // variable cachée pour l'utilisateur
};
int main(array<System::String ^> ^args)
{
  MyClass obj(10);
  for(int i=-1;i<12;i++) obj.nIntArray[i]=i*2;
  for(int i=0;i<obj.GetSizeArray();i++) Console
  ::WriteLine((i).ToString()+" :="+obj.nIntArray[i].ToString());
  return 0;
}

```

### III-1.4 - Les propriétés indexées par défaut

La propriété indexée par défaut permet d'appliquer l'indexation directement sur l'instance de la classe à laquelle appartient l'objet.

La syntaxe est la même que pour les propriétés indexées, il faudra juste rajouter le mot clef default en lieu et place du nom de la propriété.

#### Exemple:

```

ref class IndexedProperty
{
public:
  IndexedProperty()
  {
    m_array= gnew array<int>{1,2,3,10,100,200};
  }
  property int ^ default [int]
  {
    int ^get(int index)
    {
      if(index<0) index=0;
      else if(index>m_array->Length) index=m_array->Length-1;
      return m_array[index];
    }
  }
}

```

```
    }  
    }  
private:  
    array<int>^ m_array;  
};  
  
int main(array<System::String ^> ^args)  
{  
    IndexedProperty TestIndex;  
  
    Console::WriteLine(TestIndex[-10]);  
    Console::WriteLine(TestIndex[2]);  
    Console::WriteLine(TestIndex[10]);  
    return 0;  
}
```

### III-J - Gestion des transtypages entres classes

De même qu'en C++, les opérations de transtypage d'une classe à l'autre sont possibles.

Néanmoins le transtypage avec **static\_cast** du C++ ne peut être vérifié et doit être considéré comme peu sûr (unsafe) On lui préférera le mot clef **safe\_cast**.

Quand à **dynamic\_cast** il fonctionne comme en C++ et renverra nul si le transtypage n'est pas correct.

### III-K - Les différents types de classes

#### III-K.1 - Les classes ref sealed

Une classe **ref sealed** (scellée) ne peut être héritée.

Il faut vraiment avoir un motif particulier pour empêcher l'héritage sur une classe ref.

```
ref class MySealClass sealed  
{  
};  
ref class DerivedClass: public MySealClass //error C3246: 'DerivedClass' : ne peut pas hériter de  
'MySealClass',  
{  
    // car il a été déclaré comme 'sealed'  
};
```

#### III-K.2 - Les classes ref Abstraites

Une classe ref abstraite se construit comme une classe ref classique sans son implémentation.

Elle doit posséder au minimum une fonction virtuelle pure.

Elle peut posséder des variables, des méthodes, des propriétés, des constructeurs et des destructeurs.

Mais elle ne pourra pas être instanciée directement, seule une classe héritée de cette classe pourra être instanciée.

**Exemple :**

```
ref class AbstractClass abstract
{
public:
    AbstractClass()
    {
        m_n=100;
    }
    virtual void function()=0;
    void GetNumber()
    {
        Console::WriteLine(m_n.ToString());
    }
private:
    int m_n;
};
```

### III-K.3 - Classe Interface

Le fonctionnement est le même que pour une classe abstraite, si ce n'est que :

L'accès doit être public, et donc par défaut c'est ce mode qui est actif.

La classe ne peut être composée que de fonctions virtuelles pures.

Donc pas de variables ou méthodes.

On pourra éviter le mot clef **virtual et =0** puisque la classe est explicitement déclarée comme une interface.

#### Exemple :

```
interface class MyInterfaceClass
{
    void Function();
};
ref class MyClass
{
    void OneFunction(){Console::WriteLine("OneFunction");}
};
ref class MyFinalClass: public MyClass, public MyInterfaceClass
{
public:
    void Function(){Console::WriteLine("Function");}
};
```

### III-K.4 - Les classes génériques

Elle sont similaires aux templates du C++.

#### Quelles sont les différences ? :

Les spécialisations ne sont pas autorisées et il n'y pas de paramètre par défaut.

Elles disposent d'un système de contrainte du type.

Elles ont un runtime, les objets génériques sont vérifiables.

### Autres différences importantes :

Pour un template tout se passe à la compilation du code, pour une classe générique c'est à l'exécution..

Une classe générique pourra être utilisée par les autres langages .net un template non .

### Exemples:

On pourrait être tenté par ce type de code

```
generic<typename T>
T Min(T a,T b)
{
    return(a<b?a:b); // error C2676
}
```

Ce code provoque l'erreur de compilation **C2676** qui me dit que le type T ne supporte pas l'opérateur binaire "<" #

Après recherche, une fonction ou classe générique ne supporte pas l'utilisation des opérateurs sur le paramètre .

C'est une limitation dont il faudra tenir compte.

 *Il semble que certaines versions du compilateur supportent ce type de code, c'est une bizarrerie que j'ai constatée avec mon collègue Nico-Pyright(c).*

*Mes recherches sur le sujet m'ont amenées à une discussion sur le forum MSDN où Herb Sutter lui-même disait que cette fonctionnalité n'était pas supportée dans les langages .Net, d'ailleurs en C# ce code ne compile pas.*

*A suivre#*

La déclaration d'une classe générique se fait de la même manière qu'une classe template:

```
generic <class T>
ref class Point
{
public:
    Point(T x,T y){xCoord=x;yCoord=y;}
    void ShowCoord()
    {
        Console::WriteLine("XCoord:"+xCoord->ToString());
        Console::WriteLine("YCoord:"+yCoord->ToString());
    }
    static bool operator ==(Point ^lefth,Point ^righth)
    {
        // conversion en string ,ce n'est pas terrible c'est juste pour l'illustration...
        String ^lx=lefth->xCoord->ToString();
        String ^rx=righth->xCoord->ToString();
        String ^ly=lefth->yCoord->ToString();
        String ^ry=righth->yCoord->ToString();
    }
}
```

```

    return (lx==rx && ly==ry);
}
static bool operator ==(Point %left,Point %right)
{
    // conversion en string ,ce n'est pas terrible c'est juste pour l'illustration...
    String ^lx=left.xCoord->ToString();
    String ^rx=right.xCoord->ToString();
    String ^ly=left.yCoord->ToString();
    String ^ry=right.yCoord->ToString();

    return (lx==rx && ly==ry);
}
property T xCoord;
property T yCoord;
};
int main(array<System::String ^> ^args)
{
    Point<int>^ pt= gnew Point<int>(10,100);
    pt->ShowCoord();
    Point<int>^ pt2= gnew Point<int>(10,100);
    Console::WriteLine("pt==pt2 "+(pt==pt2).ToString());

    Point<int> pt3(10,100);
    pt3.ShowCoord();
    Point<int> pt4(10,100);
    Console::WriteLine("pt3==pt4 "+((pt3)==(pt4)).ToString());
    return 0;
}

```

Dans cet exemple, j'ai contourné le problème lié à l'utilisation d'un opérateur sur le type en paramètre, en le convertissant en String (pour l'instant) pour réaliser le test.

### III-K.4.a - Définition d'une contrainte de type

Les contraintes permettent de spécifier le type accepté en paramètre.

La syntaxe générale est la suivante :

**where typeParametre : [class contrainte,] [interface liste des contraintes]**

La contrainte peut être une classe ou une interface comme **IComparable** ou **IEnumerable**.

**Exemples d'écritures issus de MSDN :**

**Une contrainte avec une interface :**

```

interface class IItem {};

generic <class ItemType> where ItemType : IItem
ref class Stack
{
};

```

**Une contrainte avec une classe :**

```
generic <typename T>
ref class G1 {};

generic <typename Type1, typename Type2> where Type1 : G1<Type2> // OK, G1 takes one type
parameter
ref class G2
{
};
```

Pour montrer l'utilisation d'une contrainte de type j'ai repris l'exemple précédent de la classe générique Point.

```
generic <class T> where T: IComparable
ref class Point
{
public:
Point(T x,T y){xCoord=x;yCoord=y;}
void ShowCoord()
{
Console::WriteLine("XCoord:"+xCoord->ToString());
Console::WriteLine("YCoord:"+yCoord->ToString());
}

static bool operator ==( Point ^lefth, Point ^righth)
{
return !lefth->xCoord->CompareTo(righth->xCoord) &&
!lefth->yCoord->CompareTo(righth->yCoord);
}

static bool operator ==(Point %left,Point %right)
{
return !left.xCoord->CompareTo(right.xCoord) &&
!left.yCoord->CompareTo(right.yCoord);
}

property T xCoord;
property T yCoord;
};

int main(array<System::String ^> ^args)
{
Point< int >^ pt= gnew Point< Int32>(10,100); // j'ai utilisé un int classique pourquoi ça
fonctionne ?
pt->ShowCoord();
Point< Int32 >^ pt2= gnew Point< Int32 >(10,100);
Console::WriteLine("pt==pt2 "+(pt==pt2).ToString());

Point< Int32 > pt3(10,100);
pt3.ShowCoord();
Point< Int32> pt4(10,100);
Console::WriteLine("pt3==pt4 "+((pt3)==(pt4)).ToString());
return 0;
}
```

j'ai spécifié que l'argument **T** devait hériter de l'interface **IComparable**.

Si vous rappelez de ce qui a été dit sur les types de données: ils disposent tous d'alias dans le framework.

Ainsi maintenant je peux utiliser un **int** ou son alias **Int32** définit dans l'espace de nom **Systeme**, pourquoi ?

Ces alias sont des **value class** héritées (entre autre ) de l'interface **IComparable**.

la difficulté rencontrée précédemment n'en est plus une....

Enfin notre fonction générique **Min** pourra donc s'écrire comme suit:

```
generic<class T> where T: IComparable
T Min(T a,T b)
{
    return(a->CompareTo(b)<0?a:b);
}
int main(array<System::String ^> ^args)
{
    int a=20;
    int b=10;
    Console::WriteLine(Min(a ,b).ToString());

    Console::WriteLine(Min<int>(20,10).ToString());

    return 0;
}
```

### III-K-5 - Les Templates

Il n'y pas de différences avec le C++ si ce n'est la possibilité d'utiliser une classe référence, On pourra donc définir classiquement une fonction template :

```
template<typename T>
T Min(T a,T b)
{
    return(a<b?a:b);
}
int main(array<System::String ^> ^args)
{
    int n=10;
    int n2=20;
    Console::WriteLine(Min(n,n2));
}
```

Et déclarer une ref class template :

```
template <class T>
ref class Point
{
public:
    Point(T x,T y){xCoord=x;yCoord=y;}
    void ShowCoord()
    {
        Console::WriteLine("XCoord:"+xCoord.ToString());
        Console::WriteLine("YCoord:"+yCoord.ToString());
    }
    static bool operator ==(Point ^lefth,Point ^righth)
    {
        return (lefth->xCoord==righth->xCoord && lefth->yCoord==righth->yCoord);
    }
    static bool operator ==(Point %left,Point %right)
    {
        return (left.xCoord==right.xCoord && left.yCoord==right.yCoord);
    }
    property T xCoord;
    property T yCoord;
};
int main(array<System::String ^> ^args)
{
```

```
Point<int>^ pt= gnew Point<int>(10,100);
pt->ShowCoord();
Point<int>^ pt2= gnew Point<int>(10,100);
Console::WriteLine("pt==pt2 "+(pt==pt2).ToString());

Point<int> pt3(10,100);
pt3.ShowCoord();
Point<int> pt4(10,100);
Console::WriteLine("pt3==pt4 "+(pt3==pt4).ToString());
    return 0;
}
```

Vous remarquerez dans cet exemple:

L'utilisation des propriétés, et la surcharge de l'opérateur == supportant les deux formes de création de l'objet: par handle ou sur la pile.

De même qu'en C++ notre classe ref template supportera la spécialisation complète ou partielle de la classe.

## III-L - Les Exceptions

Elles se codent de la même manière qu'en C++ avec un bloc " try catch ".

```
try
{
// votre code qui doit être contrôlé
}
catch(ExceptionType e)
{
// code d'interception
}
```

On pourra à la suite du **catch** rajouter d'autres **catch** pour d'autres interceptions spécifiques.

### III-L.1 - Exception spécialisée

Le Framework possède dans l'espace de noms **Systeme** une exception **InvalidCastException** qui pourra être utilisée conjointement avec l'opérateur **safe\_cast**.

L'exemple qui suit est une adaptation d'un exemple C# trouvé sur MSDN :

```
ref class Employee
{
public :
    static void PromoteEmployee(Object ^emp)
    {
//Cast object to Employee.
Employee ^e = (Employee ^) emp;
// Increment employee level.
e->EmLevel = e->EmLevel + 1;
}
    property int EmLevel;
};
try
{
Object ^o = gnew Employee;
```

```
DateTime ^newyears = gnew DateTime(2001, 1, 1);  
//Promote the new employee.  
Employee::PromoteEmployee(o);  
//Promote DateTime; results in InvalidCastException as newyears is not an employee instance.  
Employee::PromoteEmployee(newyears);  
}  
catch (InvalidCastException ^e)  
{  
    Console::WriteLine("Error passing data to PromoteEmployee method. " + e);  
}
```

Dans cet exemple j'ai volontairement employé le cast du C qui provoquera la levée de l'exception sur la deuxième tentative de promotion.

Si j'avais utilisé l'opérateur **static\_cast** l'exception **ArgumentException** aurait été levée.

Si j'avais utilisé l'opérateur **safe\_cast** j'aurais eu le même résultat qu'avec le cast C, ceci est dû au fait que le cast C utilise en interne **safe\_cast**.

Le paradoxe n'aura pas échappé au programmeur C++ utilisant **static\_cast** dans ses sources au lieu du cast C, en C++/CLI il faudra mettre de côté cette bonne habitude et utiliser **safe\_cast** ou revenir au bon vieux cast du C.

### III-L.2 - La classe de base Exception du framework

Le Framework .net dispose d'une classe de base pour la gestion des exceptions : **Exception**.

Deux types d'exceptions sont gérés :

Les exceptions d'application: **ApplicationException**

Cette classe de base permettra de gérer nos propres exceptions.

Les exceptions systèmes : **SystemException**.

Elles gèrent toutes les exceptions d'entrées/sorties etc..

#### III-L.2.a - Interception des exceptions utilisateurs

J'ai repris l'exemple précédent en le modifiant pour lever ma propre exception :

```
ref class EmployeeException : public ApplicationException  
{  
public:  
    EmployeeException(System::String ^err):ApplicationException(err)  
    {  
    }  
};  
  
ref class Employee  
{  
public :  
    static void PromoteEmployee(Object ^emp)  
    {  
    }  
}
```

```

Employee ^e;
//Cast object to Employee.
try
{
e= safe_cast<Employee ^> (emp);
}
catch(InvalidCastException ^err)
{
throw gnew EmployeeException("\nErreur de conversion sur le type Employé "+err);
return;
}
// Increment employee level.
e->EmLevel = e->EmLevel + 1;
}
property int EmLevel;
};
int main(array<System::String ^> ^args)
{
Object ^o = gnew Employee;
DateTime ^newyears = gnew DateTime(2001, 1, 1);
//Promote the new employee.
Employee::PromoteEmployee(o);
//Promote DateTime; results in InvalidCastException as newyears is not an employee instance.
Employee::PromoteEmployee(newyears);
return 0;
}

```

## III-M - Delegates

Un delegate est une classe managée (ref class) qui permet d'appeler une méthode qui partage la même signature qu'une fonction globale ou qu'une classe possédant des méthodes avec cette même signature.

Le framework supporte deux formes de delegates :

**System::Delegate** un delegate qui accepte d'appeler uniquement une méthode.

**System::MulticastDelegate** : un delegate qui accepte d'appeler une chaine de méthodes.

Les méthodes utilisées pour le delegate peuvent être :

Globale (comme en C), une méthode statique à une classe, une méthode d'une instance.

**Exemple** : Déclaration du delegate :

```
delegate void FuncMessDelegate(String ^mess);
```

Maintenant les trois formes d'utilisation :

```

void GlobalMess(String ^mess)
{
Console::Write("Fonction Globale: ");
Console::WriteLine(mess);
}
ref class OneClass

```

```
{
public:
    static void staticMethodeMess(String ^mess)
    {
        Console::Write("Fonction statique: ");
        Console::WriteLine(mess);
    }
};
ref class AnotherClass
{
public:
    void MethodeMess(String ^mess)
    {
        Console::Write("Fonction d'une instance: ");
        Console::WriteLine(mess);
    }
};
```

L'appel de la fonction :

```
int main(array<System::String ^> ^args)
{
    // declaration du delegate
    FuncMessDelegate ^Global= gcnew FuncMessDelegate(&GlobalMess);
    // ajouter une fonction au delegate
    FuncMessDelegate ^statique= gcnew FuncMessDelegate(&OneClass::staticMethodeMess);

    AnotherClass ^pAnotherClass= gcnew AnotherClass;

    FuncMessDelegate ^instance= gcnew FuncMessDelegate(pAnotherClass, &AnotherClass::MethodeMess);

    // l'appel de la fonction#
    Global->Invoke("Hello");

    statique->Invoke("Hello");
    instance->Invoke("Hello");

    return 0;
}
```

Les delegates peuvent être combinés sous forme de chaîne et un élément peut être supprimé.

On utilisera la méthode **Combine()** ou l'opérateur "+" pour l'ajout.

Et la méthode **Remove()** ou l'opérateur "-" pour la suppression.

**Exemple :**

```
FuncMessDelegate ^ChaineMess= gcnew FuncMessDelegate(&GlobalMess);
ChaineMess += gcnew FuncMessDelegate(&OneClass::staticMethodeMess);

AnotherClass ^pAnotherClass= gcnew AnotherClass;
ChaineMess += gcnew FuncMessDelegate(pAnotherClass, &AnotherClass::MethodeMess);

ChaineMess->Invoke("Hello");

ChaineMess -= gcnew FuncMessDelegate(&OneClass::staticMethodeMess);
ChaineMess->Invoke("Hello2");
```

**Note :**

J'ai volontairement utilisé les opérateurs à la place des méthodes car ceux-ci imposent un cast vers le delegate déclaré, ce qui alourdit grandement l'écriture.

## III-N - Events

Un **event** est une implémentation spécifique du **delegate** ou plutôt du **multicast delegate**.

Un **event** permet à partir d'une classe d'appeler des méthodes situées dans d'autres classes sans rien connaître de ces classes.

Il est ainsi possible pour une classe d'appeler une chaîne de méthodes issue de différentes classes.

**Exemple :**

```
using namespace System;

delegate void DelegateAnalyse(int %n); // la fonction de traitement recoit une reference sur un
entier.

// classe declencheur de traitement
ref class AnalyseTrigger
{
public:
    event DelegateAnalyse ^OnWork;
    void RunWork(int %n)
    {
        OnWork(n);
    }
};

// classe effectuant un traitement
ref class Analyse
{
public:
    AnalyseTrigger ^ m_AnalyseTrigger;
    Analyse(AnalyseTrigger ^src)
    {
        if(src==nullptr)
            throw gcnew ArgumentNullException("erreur argument non specifié");
        m_AnalyseTrigger=src;
        m_AnalyseTrigger->OnWork+= gcnew DelegateAnalyse(this,&Analyse::Treatment);
        m_AnalyseTrigger->OnWork+= gcnew DelegateAnalyse(this,&Analyse::TreatmentTwo);
    }
    void RemoveTreatmentTwo()
    {
        m_AnalyseTrigger->OnWork-=gcnew DelegateAnalyse(this,&Analyse::TreatmentTwo);
    }
    // traitement declenché
    void Treatment(int %n)
    {
        Console::Write("Class Analyse Treatment +=10 n:= ");
        Console::Write(n);
        n+=10;
        Console::Write(" -> n=");
        Console::WriteLine(n);
    }
    // traitement declenché
    void TreatmentTwo(int %n)
    {

```

```
Console::Write("Class Analyse TreatmentTwo *=2 n:= ");
Console::Write(n);
n*=2;
Console::Write(" -> n=");
Console::WriteLine(n);
}
};

// classe effectuant un traitement
ref class AnalyseTwo
{
public:
AnalyseTrigger ^m_AnalyseTrigger;

AnalyseTwo(AnalyseTrigger ^src)
{
if(src==nullptr)
throw gcnew ArgumentNullException("erreur argument non spécifié");
m_AnalyseTrigger=src;
m_AnalyseTrigger->OnWork+= gcnew DelegateAnalyse(this,&AnalyseTwo::Treatment);
}
// traitement déclenché
void Treatment(int %n)
{
Console::Write("Class AnalyseTwo Treatment +=2 n:= ");
Console::Write(n);
n+=2;
Console::Write(" -> n=");
Console::WriteLine(n);
}
};

int main(array<System::String ^> ^args)
{
//element declancheur de l'action fixée par le delegate DelegateAnalyse
AnalyseTrigger ^Trigger = gcnew AnalyseTrigger();

// classe traitements
Analyse ^analyse = gcnew Analyse(Trigger);

// classe traitements
AnalyseTwo ^analyseTwo = gcnew AnalyseTwo(Trigger);

int n=2;

Trigger->RunWork(n);
analyse->RemoveTreatmentTwo();

Console::WriteLine("-----");
n=2;
Trigger->RunWork(n);

return 0;
}
```

## IV - Conclusion

### **Pour les programmeurs C++ :**

Le C++/CLI est l'occasion d'aborder le monde .Net avec un langage familier.

C'est aussi une opportunité de faire évoluer une application existante en C++ en utilisant les outils du framework .Net,

Comme je l'ai précisé en introduction, un programme MFC peut être enrichi de Winform, et pourquoi pas utiliser une vue WPF.

Vous l'aurez compris le formidable avantage du C++/CLI c'est la récupération de votre code C++ de vos projets et la possibilité d'interopérabilité avec les autres langages .NET.

Vous voilà armés avec les éléments de base syntaxique du C++/CLI, il ne vous reste plus qu'à mettre en pratique.

J'ai volontairement mis de coté certains sujets comme les collections de données, la gestion des fichiers etc ..., qui seront prétexte à un autre article. Je vous invite à consulter la [FAQ C++/CLI](#) qui traite aussi nombre de points abordés dans cet article.

### **Pour les autres programmeurs .NET :**

Quel intérêt un programmeur C# (par exemple) peut-il avoir à regarder le C++/CLI ?

La possibilité d'interfacer son application avec du code C++ ou C existant, le tout encapsulé dans une assembly.

C'est un argument à méditer pour la récupération de code existant ou tout simplement l'utilisation d'une bibliothèque sans équivalence dans le langage .NET pratiqué.

## V - Remerciements

Je remercie toute l'équipe C++, particulièrement **Nico-pyright(c)**, l'équipe Dotnet, pour leur relecture attentive du document.

Merci à **Skalp** pour la relecture finale de cet article

