

L'environnement .NET

1. Le code managé et la machine virtuelle CLR

Comme Java, .NET fait partie de la famille des environnements virtualisés. Le compilateur C++ managé ne produit pas du code assembleur directement exécuté par le microprocesseur mais un code intermédiaire, lui-même exécuté par une machine "virtuelle", la CLR (*Common Language Runtime*). Cette couche logicielle reproduit tous les composants d'un ordinateur - mémoire, processeur, entrées-sorties...

Cette machine doit s'adapter au système d'exploitation qui l'héberge, et surtout optimiser l'exécution du code.



Il y a là une différence importante entre .NET et Java. Alors que Microsoft a évidemment fait le choix de se concentrer sur la plate-forme Windows, d'autres éditeurs ont porté Java sur leurs OS respectifs - Unix, AIX, Mac OS, Linux... La société Novell a cependant réussi le portage de .NET sur Linux, c'est le projet Mono.

Cet environnement virtualisé a une conséquence très importante sur les langages supportés ; les types de données et les mécanismes objets sont ceux de la CLR. Comme Microsoft était dans une démarche d'unification de ses environnements, plusieurs langages se sont retrouvés éligibles à la CLR, quitte à adapter un peu leur définition. Le projet de Sun était plutôt une création ex nihilo, et de ce fait seul le langage Java a réellement été promu sur la machine virtuelle JVM.

2. Les adaptations du langage C++ CLI

Le terme CLI signifie *Common Language Infrastructure* ; c'est l'ensemble des adaptations nécessaires au fonctionnement de C++ pour la plate-forme .NET / CLR.

a. La norme CTS

Le premier changement concerne les types de données. En successeur de C, le C++ standard a basé sa typologie sur le mot machine (`int`) et le caractère de 8 bits (`char`). Confronté à des problèmes de standardisation et d'interopérabilité, Microsoft a choisi d'unifier les types de données entre ses différents langages.

Les types primitifs, destinés à être employés comme variables locales (boucles, algorithmes...), sont des types "valeurs". Leur zone naturelle de stockage est la pile (*stack*), et leur définition appartient à l'espace de noms **system** :

| | | |
|----------------------------|------------------------------|--------------------------|
| <code>wchar_t</code> | <code>System::Char</code> | Caractère unicode |
| <code>signed char</code> | <code>System::SByte</code> | Octet signé |
| <code>unsigned char</code> | <code>System::Byte</code> | Octet non signé |
| <code>double</code> | <code>System::Double</code> | Décimal double précision |
| <code>float</code> | <code>System::Float</code> | Décimal simple précision |
| <code>int</code> | <code>System::Int32</code> | Entier 32 bits |
| <code>long</code> | <code>System::Int64</code> | Entier 64 bits |
| <code>unsigned int</code> | <code>System::UInt32</code> | Entier 32 bits non signé |
| <code>unsigned long</code> | <code>System::UInt64</code> | Entier 64 bits non signé |
| <code>short</code> | <code>System::Int16</code> | Entier court 16 bits |
| <code>bool</code> | <code>System::Boolean</code> | Booléen |
| <code>void</code> | <code>System::Void</code> | Procédure |

Voici un exemple utilisant à la fois la syntaxe habituelle de C++ et la version "longue" pour définir des nombres entiers. Cette dernière ne sera utilisée qu'en cas d'ambiguïté, mais il s'agit en fait de la même chose.

```
int entier = 4; // alias de System::Int32
System::Int32 nombre = entier;
```

Les structures managées (**ref struct**) sont composées de champs reprenant les types ci-dessus. Elles sont créées sur la pile et ne sont pas héritables.

```
ref struct Point
{
    public:
        int x,y;
};

int main(array<System::String ^> ^args)
{
    Point p; // initialisation automatique sur la pile : pas de gcnew
    p.x = 2;
    p.y = 4;

    return 0;
}
```

Comme tous les types valeurs, les structures managées n'autorisent pas l'effet de bord à moins qu'une référence explicite n'ait été passée (voir ci-dessous les références suivies).

Par opposition les classes managées (**ref class**) sont créées sur le tas et correspondent davantage au fonctionnement des classes (structure) du C/C++ standard. Évidemment, elles sont héritables, manipulées par références, et instanciées par l'opérateur **gcnew**.

```
ref class Personne
{
    public:
        String^ nom;
        int age;
};

int main(array<System::String ^> ^args)
```

```

{
    Personne^ pers = gnew Personne();
    pers->nom = "Marc";
    pers->age = 35;

    return 0;
}

```

Naturellement, les structures et les classes managées supportent la définition de méthodes, C++ est bien un langage objet !

Les langages .NET sont fortement typés, en évitant autant que possible la généricité "fourre-tout" du `void*` issu du C, ou du `Variant` de VB. En opposition, la norme CTS prévoit que l'ensemble des types de données (valeurs ou référence) héritent d'un comportement commun `System::Object`. Nous découvrirons un peu plus loin comment cette caractéristique est exploitée dans les classes de collections d'objets.

b. La classe `System::String`

Parmi les types intrinsèques à .NET on trouve la classe `System::String` pour représenter les chaînes. Elle n'est pas aussi intégrée au langage C++ CLI que dans C#, mais cependant elle offre exactement les mêmes services. Microsoft l'a de plus aménagée pour faciliter la manipulation et la conversion avec `char*` :

Voici un premier exemple de construction de chaînes. On remarquera l'emploi facultatif du symbole `L` devant les littérales de chaîne, ainsi que le symbole `^` dont la signification sera expliquée un peu plus tard.

```

char* c = "Bonjour C++";
String ^ chaine1 = gnew String(c); // construction à partir d'un char*
String ^ chaine2 = L"C++ CLI c'est formidable"; // littérale de chaîne .NET
String ^ chaine3 = "Un univers à découvrir"; // idem

Console::WriteLine(chaine1);
Console::WriteLine(chaine2 + ". " + chaine3);

```

```

C:\WINDOWS\system32\cmd.exe
Bonjour C++
C++ CLI c'est formidable. Un univers à découvrir
Appuyez sur une touche pour continuer... _

```

Nous poursuivons avec quelques exemples d'emploi de méthodes et de propriétés de la classe `System::String`. Cet extrait de code emploie une nouvelle instruction, `for each`, qui réalise une itération sur l'ensemble des éléments d'une collection. Comme dans le cas de la STL, une chaîne est assimilée à une collection de caractères :

```

// manipuler les chaînes
String ^ prefixe = "Bon";
if(chaine1->StartsWith(prefixe))
    Console::WriteLine("chaîne1 commence par {0}", prefixe);

// itérer sur les caractères avec une boucle for each
for each(wchar_t c in chaine1)
{
    Console::Write(" " + c);
}
Console::WriteLine();

// itérer classiquement avec un for
for(int i=0; i<chaine2->Length; i++)
    Console::Write( chaine2[i] );

Console::WriteLine();

```

```

C:\WINDOWS\system32\cmd.exe
chaîne1 commence par Bon
B o n j o u r C + +
C++ CLI c'est formidable
Appuyez sur une touche pour continuer... _

```

La classe `System::String` contient de nombreuses méthodes et propriétés que tout programmeur a intérêt à découvrir. Beaucoup d'entre elles se retrouvent également dans la classe `string` de la STL.

Voici maintenant comment obtenir un `char*` à partir d'un `String`. L'opération repose sur le concept de marshalling, c'est-à-dire de mise en rang de l'information. Pourquoi est-ce si difficile ? Les `char*` n'ont pas beaucoup d'intérêt dans le monde .NET qui propose des mécanismes plus évolués. Mais ces mêmes `char*` sont indispensables dans le monde Win32 où un très grand nombre d'éléments de l'API les utilisent. Le passage d'un monde à l'autre, la transformation des données, s'appelle le marshalling. C'est donc la classe `System::Runtime::InteropServices::Marshal` qui est chargée de "convertir" un `String` en `char*` :

```
// conversion vers char* (utile pour les accès natifs à Win32)
char* cetoile =
static_cast<char*>(Marshal::StringToHGlobalAnsi(chaine1).ToPointer());

// utilisation de la chaîne
// ...

// libérer le buffer
Marshal::FreeHGlobal(safe_cast<IntPtr>(cetoile));
```

Nous constatons que le programmeur est responsable d'allouer et de libérer le buffer contenant la chaîne exprimée en `char*`. La présentation du garbage collector va donner des informations complémentaires à ce sujet.

c. Le garbage collector

Une caractéristique des environnements virtualisés comme Java ou .NET est l'emploi d'un ramasse-miettes ou garbage collector en anglais. Il s'agit d'un dispositif de recyclage et de compactage automatique de la mémoire. Autant le fonctionnement de la pile est assez simple à systématiser, autant les algorithmes de gestion du tas diffèrent d'un programme à l'autre. Or, les langages comme le C et le C++ mettent en avant le mécanisme des pointeurs qui supposent une totale liberté dans l'adressage de la mémoire.

Quel est le principal bénéfice de la fonction garbage collector ? En ramassant les miettes, on évite le morcellement de la mémoire et de ce fait on optimise son emploi et sa disponibilité. Ceci est pourtant difficilement compatible avec l'emploi des pointeurs et leur arithmétique qui ne peut être prédite à la compilation. Les langages nouvellement créés pour les environnements virtualisés comme Java et C# ont donc fait "disparaître" la notion

de pointeur au profit des seules références, puisqu'une référence n'est en principe pas "convertible" en adresse mémoire.



C# comme C++ CLI disposent malgré tout des pointeurs, mais leur utilisation est strictement encadrée pour ne pas perturber le fonctionnement du garbage collector.

Le ramasse-miettes est activé périodiquement par la CLR lorsque la mémoire vient à manquer ou qu'une allocation conséquente est sollicitée. Le programmeur peut également anticiper et demander lui-même son déclenchement. Le principe est de compter le nombre de références actives sur les zones mémoire allouées et de libérer toutes les zones qui ne sont plus référencées. Dans un second temps, le garbage collector (GC) peut décider de déplacer des blocs mémoire pour compacter les espaces alloués et rendre disponibles de plus grands blocs. Les références correspondantes sont actualisées avec les nouvelles adresses, ce qui a pour conséquence de rendre les pointeurs inutilisables.

Mémoire avant le passage du GC

| Référence | Adresse | Contenu |
|-----------|---------|-----------|
| REF 4 | 0x5000 | Objet 2 |
| REF 3 | 0x4000 | Tableau 2 |
| | 0x3000 | Libre |
| REF 2 | 0x2000 | Tableau 1 |
| REF 1 | 0x1000 | Objet 1 |

Mémoire après le passage du GC

| Référence | Adresse | Contenu |
|-----------|---------|-----------|
| | 0x5000 | Libre |
| | 0x4000 | Libre |
| REF 4 | 0x3000 | Objet 2 |
| REF 2 | 0x2000 | Tableau 1 |
| REF 1 | 0x1000 | Objet 1 |

Activation du GC

Comptage et suivi des références

| Référence | Variables | Compte |
|-----------|---------------------------|--------|
| REF 1 | objet_1 | 1 |
| REF 2 | tab_a, tab_b | 2 |
| REF 3 | tab_c (hors de portée) | 0 |
| REF 4 | objet_2 | 1 |

Comptage et suivi des références

| Référence | Variables | Compte |
|-----------|-----------------|--------|
| REF 1 | objet_1 | 1 |
| REF 2 | tab_a, tab_b | 2 |
| REF 4 | objet_2 | 1 |

Fonctionnement du ramasse-miettes

d. Construction et destruction d'objets

Le langage C++ CLI offre deux visages. Le premier est une gestion classique de la mémoire sur un tas spécifique, avec constructeurs et destructeurs invoqués par les opérateurs `new` et `delete` (ou automatiquement si l'objet est instancié sur la pile). Le second repose sur la gestion managée de la mémoire, avec une utilisation de références sur le tas du CLR rendant possible l'usage du garbage collector.

C'est au moment de la définition de la classe que l'on opte pour l'un ou l'autre des modes. Si la classe est définie à l'aide du mot-clé `ref`, il s'agit du mode managé. Autrement, c'est le schéma classique.

Étudions la classe suivante :

```

// la présence du mot-clé ref indique
// qu'il s'agit d'une classe managée
ref class Personne
{
public:

String ^ nom; // référence vers un objet managé de type String
int age;

    Personne()
    {
        nom = nullptr; // nullptr est une référence managée et non
                        // une adresse
        age = 0;
    }

    Personne(String ^ nom, int age)
    {
        // this est donc une référence managée et non un pointeur
        this->nom = nom;
        this->age = age;
    }
};

```

Le mot-clé **nullptr** représente une référence managée et non une adresse. Dans l'univers classique, **NULL** (littéralement `(void*)0`) représente une adresse qui ne peut être utilisée. Dans le cas des classes managées, c'est une constante dont le programmeur n'a pas à connaître la valeur puisqu'il n'y a pas d'arithmétique des pointeurs. Par commodité, **this** devient l'autoréférence de la classe et n'est pas un pointeur. Cependant, Microsoft a choisi l'opérateur `->` pour accéder aux membres à partir d'une référence managée afin de ne pas troubler les programmeurs expérimentés dans le C++ classique.

Nous aurons également remarqué l'utilisation du symbole **^** (*caret*) qui désigne la référence vers un objet managé. On parle aussi de handle d'objet à cet effet. Voici à présent comment on procède pour instancier une classe managée :

```

int main(array<System::String ^> ^args)
{

```

```

// déclaration d'une référence
Personne ^ objet1; // une référence vers un objet managé

// instanciation et invocation du constructeur
objet1 = gcnew Personne("Corinne", 25);

// utilisation de l'objet
Console::WriteLine("objet1: nom={0} age={1}",
    objet1->nom, objet1->age);

// suppression explicite de la référence
objet1 = nullptr;

return 0;
}

```

En fait, dans notre cas la ligne `objet1 = nullptr` est facultative, car la variable `objet1` est automatiquement détruite en quittant la fonction `main()`. Le garbage collector est systématiquement avisé que le compte de référence associée à `objet1` doit être diminué d'une unité. Si ce compte passe à zéro, alors le garbage collector sait qu'il peut détruire l'objet et libérer la mémoire correspondante.

En conséquence de l'emploi du ramasse-miettes, le programmeur ne libère pas lui-même ses objets puisque c'est le dispositif automatique qui s'en charge. Il y a donc une différence fondamentale dans l'approche pour libérer les ressources demandées par un objet. Comme Java, les concepteurs de C++ CLI ont choisi le terme de finaliseur, sorte de destructeur asynchrone.

Le langage C++ CLI propose les deux syntaxes, le destructeur (noté `~`) et le finaliseur (noté `!`).

```

ref class Personne
{
public:

    String ^ nom;
    int age;
}

```

```

...

~Personne()
{
    Console::WriteLine("Destructeur");
}

!Personne()
{
    Console::WriteLine("Finaliseur");
}
};

```

Le destructeur sera invoqué seulement si le programmeur détruit explicitement l'objet au moyen de l'instruction `delete`.

```

int main(array<System::String ^> ^args)
{
    // déclaration d'une référence
    Personne ^ objet1; // une référence vers un objet managé

    // instanciation et invocation du constructeur
    objet1 = gcnew Personne("Corinne", 25);

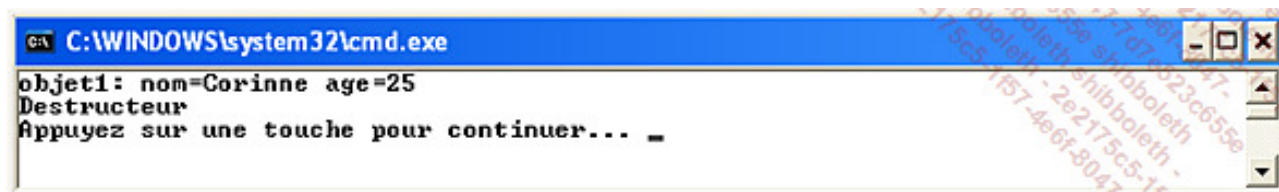
    // utilisation de l'objet
    Console::WriteLine("objet1: nom={0} age={1}",
        objet1->nom, objet1->age);

    // destruction explicite de l'objet
    // -> seul le destructeur est appelé
    // delete objet1;

    // suppression explicite de la référence
    objet1 = nullptr;

    return 0;
}

```



```
C:\WINDOWS\system32\cmd.exe
objet1: nom=Corinne age=25
Destructeur
Appuyez sur une touche pour continuer... _
```

Le finaliseur sera invoqué si l'objet est détruit par le garbage collector (ce qui dans notre cas se produit au moment où la variable `objet1` disparaît de la portée de la fonction) et à condition que le destructeur n'ait pas été invoqué.

```
int main(array<System::String ^> ^args)
{
    // déclaration d'une référence
    Personne ^ objet1; // une référence vers un objet managé

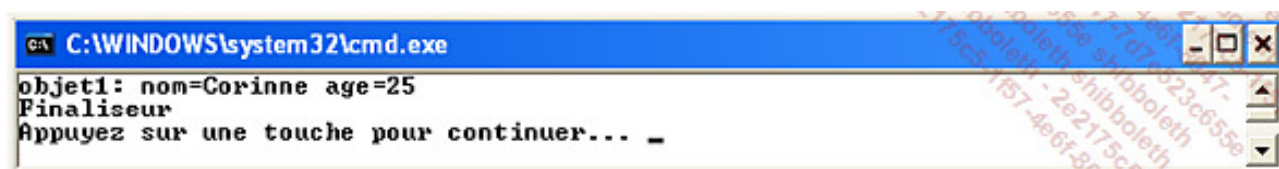
    // instanciation et invocation du constructeur
    objet1 = gcnew Personne("Corinne", 25);

    // utilisation de l'objet
    Console::WriteLine("objet1: nom={0} age={1}",
        objet1->nom, objet1->age);

    // pas de destruction explicite de l'objet
    // delete objet1;

    // suppression explicite de la référence
    objet1 = nullptr;

    return 0;
}
```



```
C:\WINDOWS\system32\cmd.exe
objet1: nom=Corinne age=25
Finaliseur
Appuyez sur une touche pour continuer... _
```

e. La référence de suivi % et le handle ^

À présent, concentrons-nous sur les références de suivi % (*tracking reference*) et sur le

handle `^` auquel nous allons élargir l'usage.

Nous commençons par quelques fonctions utilisant ces notations :

```
// le symbole % signifie la référence (alias)
void effet_de_bord_reference(int %nombre, int delta)
{
    nombre += delta;
}

// les handles sont comme des "pointeurs" pour les valeurs
int calcul_par_reference(int ^r1, int ^r2)
{
    int v1 = *r1; // déréférencement
    int v2 = *r2; // déréférencement

    return v1 + v2;
}
```

La référence de suivi `%` agit comme le symbole `&` en C++ classique. Elle crée un alias sur une variable de type valeur ou objet. Le handle `^` se comporte plutôt comme un pointeur en ce sens qu'il introduit une indirection qu'il convient d'évaluer (déréférencer).

Voyons maintenant comment sont utilisées ces fonctions :

```
// a est la référence d'un entier (et non l'entier lui-même)
int ^a = 10;

// le déréférencement * donne accès à la valeur
a = *a + 2;
effet_de_bord_reference(*a, 1); // il faut déréférencer x pour l'appel
Console::WriteLine("a = {0}", a);
```

Ce premier exemple augmente la valeur "pointée" par `x` jusqu'à 13. On remarquera l'analogie avec la notation "valeur pointée" `*` du C++ standard. La ligne la plus difficile est sans doute celle de la déclaration : `int ^ a` donne la référence d'un entier que l'on peut utiliser par `*a`. En C++ classique, l'instruction `int& r=3` n'est pas correcte, et de même que `int* p` n'alloue aucun entier.

Le second exemple est beaucoup plus proche de C++ classique :

```
// un entier créé sur la pile
int b = 20;

// une référence (alias) de la valeur b
int %c = b; // b et c désignent la même valeur

// modification "directe" de b
effet_de_bord_reference(b, 2);

// modification "directe" de c alias b
effet_de_bord_reference(c, 3);

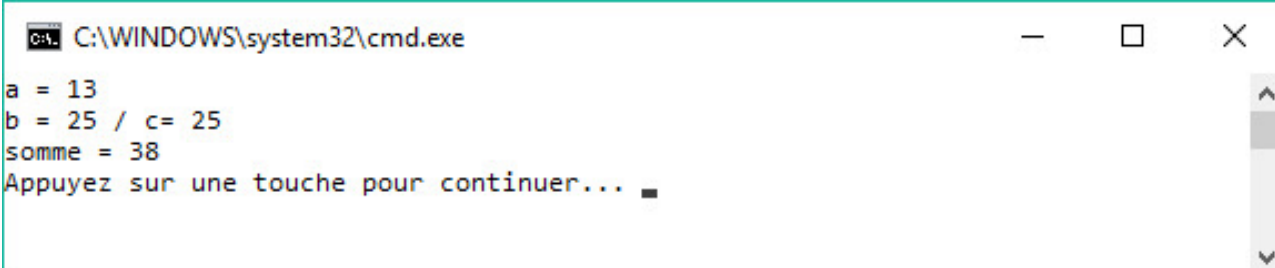
Console::WriteLine("b = {0} / c = {1}", b, c);
```

En sortie de cet exemple, `b` et `c` représentent la même valeur, soit $20 + 2 + 3 = 25$.

À l'appel d'une fonction exigeant un handle, le compilateur le fournit lui-même. Dans l'exemple qui suit, `a` est déclaré `int ^` alors que `b` est déclaré `int`.

```
int somme = calcul_par_reference(a, b);
Console::WriteLine("somme = {0} ", somme);
```

On aura remarqué dans l'implémentation de cette fonction qu'il convient de déréférencer chaque paramètre handle pour accéder à la valeur.



```
C:\WINDOWS\system32\cmd.exe
a = 13
b = 25 / c= 25
somme = 38
Appuyez sur une touche pour continuer...
```

Considérons maintenant la classe `Complexe` et deux fonctions :

```

ref class Complexe
{
public:
    double re,im;

    Complexe()
    {
        re = im = 0;
    }
};

// le handle vers un objet donne l'accès aux champs
void effet_de_bord_objet(Complexe ^ comp)
{
    comp->re = 5; // -> agit comme un déréférencement
    comp->im = 2;
}

// la référence sur un handle est comme un pointeur de pointeur
void effet_de_bord_objet_ref(Complexe ^% comp)
{
    comp = gcnew Complexe();
    comp->re=10;
    comp->im=20;
}

```

La première fonction reçoit comme paramètre un handle sur un `Complexe` ; c'est suffisant pour accéder directement aux champs de l'objet. Dans ce cas, on peut considérer `->` comme un opérateur effectuant un déréférencement. En sortie de la fonction, les champs `re` et `im` valent respectivement 5 et 2.

```

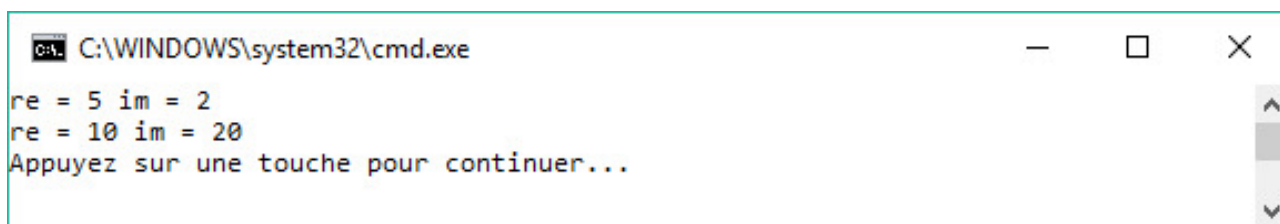
// modification par la fonction des champs de l'objet
effet_de_bord_objet(comp);
Console::WriteLine("re = {0} im = {1}",comp->re,comp->im);

```

Le paramètre de la seconde fonction est une référence de handle, sorte de pointeur de pointeur. Elle permet de substituer l'objet désigné comme paramètre par un autre :


```
// substitution par la fonction de l'objet référencé
effet_de_bord_objet_ref(comp);
Console.WriteLine("re = {0} im = {1}",comp->re,comp->im);
```

Bien qu'il soit impossible de déterminer la valeur de la référence `comp` (l'adresse de l'objet), on pourrait admettre que celle-ci a été changée par la fonction puisque c'est un autre objet qui est associé à cette référence.



```
C:\WINDOWS\system32\cmd.exe
re = 5 im = 2
re = 10 im = 20
Appuyez sur une touche pour continuer...
```

f. Le pointeur interne et les zones épinglées

De façon à limiter les modifications sur les algorithmes faisant appel aux pointeurs, Microsoft a intégré ce mécanisme dans le fonctionnement de la CLR. Un pointeur interne est l'équivalent strict d'un pointeur C++, mais sur une zone managée. On se rappelle toutefois que le GC a la faculté de déplacer des blocs mémoires, ce qui rend l'utilisation du pointeur interne un peu plus lente que les pointeurs natifs, puisque la CLR est chargée d'actualiser ces pointeurs.

La zone épinglée est une zone mémoire rendue inamovible aux yeux du GC. On gagne ainsi en rapidité puisque le pointeur épingle n'a pas à être actualisé, ceci aux dépens du morcellement de la mémoire.

Voici un premier exemple pour illustrer le fonctionnement du pointeur interne. Il s'agit d'initialiser un pointeur interne à partir de l'adresse d'un tableau de double. À titre de comparaison, nous rappelons que la syntaxe pour instancier un tableau natif est inchangée (déclaration `int* pi`). L'instanciation de tableaux managés est par contre assez différente puisqu'elle utilise l'opérateur `gcnew` et utilise la classe `System::Array`. L'initialisation du pointeur interne se pratique en demandant l'adresse du premier élément du tableau `&pd[0]`.

Les opérations d'accès par pointeur suivent les mêmes règles qu'en C++ classique sauf

qu'en arrière-plan le GC peut déplacer la mémoire sans affecter l'exécution du programme, même avec de l'arithmétique des pointeurs.

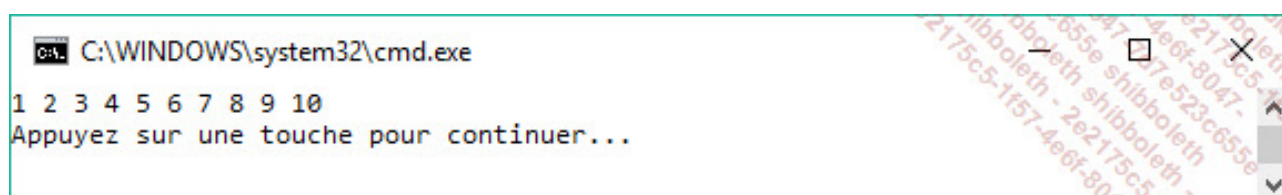
```
// un tableau de 50 entiers
// c'est un pointeur classique non managé
int* pi = new int[50];

// un tableau de 10 double
// c'est une référence (handle) managée
array<double> ^ pd = gcnew array<double>(10);
pd[0] = 1;

// un pointeur interne sur le tableau de double interior_ptr<double> pid = &pd[0];
while(++pid < &pd[0] + pd->Length)
{
    *pid = *(pid-1) + 1; // c'est beau, l'arithmétique
    // des pointeurs !
}

// vérification 1,2,3...10
for(int i=0; i< pd->Length; i++)
    Console::Write("{0} ", pd[i]);

Console::WriteLine();
```



```
C:\WINDOWS\system32\cmd.exe
1 2 3 4 5 6 7 8 9 10
Appuyez sur une touche pour continuer...
```

Dans le cas du pointeur épingle, il faut procéder en plusieurs temps. Primo, on fixe la mémoire en initialisant un pointeur épingle à partir d'une adresse. S'il s'agit d'un champ, c'est tout l'objet qui se trouve fixé en mémoire. Secundo, un pointeur natif est instancié à partir du pointeur épingle et les opérations par pointeurs ont lieu. Tertio, la zone est à nouveau rendue au contrôle complet du GC en affectant le pointeur épingle à une autre référence ou bien à `nullptr`.

```

// déclarer un pointeur épinglé fixe la mémoire correspondante
pin_ptr<double> pin = &pd[0];

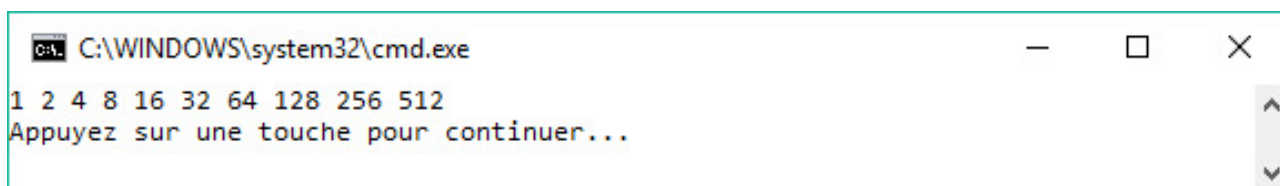
// obtention d'un pointeur "natif" pendant la durée de la manipulation
double * manip = pin;

while(++manip < &pd[0] + pd->Length)
{
    *manip = *(manip-1) * 2; // c'est beau, l'arithmétique
    // des pointeurs !
}

// vérification 1,2,4,8...
for(int i=0; i< pd->Length; i++)
    Console::Write("{0} ", pd[i] );

// libération de la zone : elle peut à nouveau bouger
pin = nullptr;

```



```

C:\WINDOWS\system32\cmd.exe
1 2 4 8 16 32 64 128 256 512
Appuyez sur une touche pour continuer...

```

g. Les tableaux et les fonctions à nombre variable d'arguments

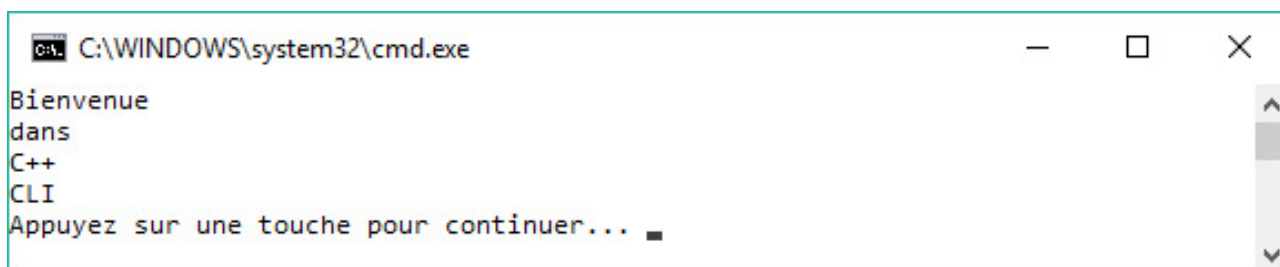
Les tableaux managés se déclarent et s'instancient à l'aide d'une syntaxe particulière. En effet, l'une des limitations des tableaux C++ classiques reste leur très grande similitude avec les pointeurs, ce qui rend parfois leur emploi risqué. La plate-forme .NET leur confère des propriétés très utiles comme la longueur (`Length`) ou l'énumération. Devenus indépendants des pointeurs, ils sont également des objets "comme les autres" que le GC peut déplacer au gré des compactages de la mémoire.

Voici un premier exemple dans lequel une chaîne est découpée autour de séparateurs - espace ou point-virgule - et les mots affichés les uns après les autres au moyen d'une boucle `for each`.

```
// un tableau de char
array<wchar_t> ^ sep = { ' ', ' ', ' ' };

// un tableau de chaînes
String^ phrase = "Bienvenue dans C++ CLI";
array<String^> ^ tabs = phrase->Split(sep);

// parcourir le tableau
for each(String^ mot in tabs)
    Console::WriteLine(mot);
```



```
C:\WINDOWS\system32\cmd.exe
Bienvenue
dans
C++
CLI
Appuyez sur une touche pour continuer...
```

L'instanciation par `gcnew` que nous avons utilisé précédemment, permet de préciser la taille du tableau, mais aussi ses dimensions. Ici une matrice de 3x3, en deux dimensions donc :

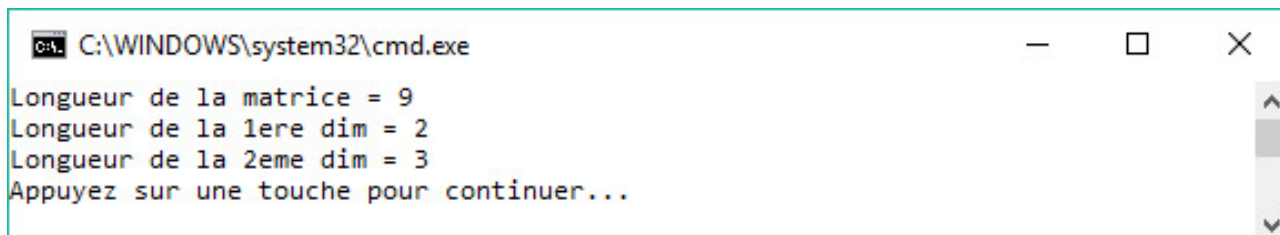
```
// un tableau à dimension double
array<float,2> ^ matrice = gcnew array<float,2>(3,3);
matrice[0,0] = 1;
matrice[0,1] = 2;
matrice[0,2] = 3;

matrice[1,0] = 4;
matrice[1,1] = 5;
matrice[1,2] = 6;

matrice[2,0] = 7;
matrice[2,1] = 8;
matrice[2,2] = 9;

// affichage des dimensions
Console::WriteLine("Longueur de la matrice = {0} ", matrice->Length);
```

```
Console::WriteLine("Longueur de 1ère dim = {0} ", matrice->GetLength(0));
Console::WriteLine("Longueur de 2ème dim = {0} ", matrice->GetLength(1));
```



```
C:\WINDOWS\system32\cmd.exe
Longueur de la matrice = 9
Longueur de la 1ere dim = 2
Longueur de la 2eme dim = 3
Appuyez sur une touche pour continuer...
```

Par voie de conséquence, les fonctions à nombre variable d'arguments de C/C++ peuvent être rendues plus sûres grâce à des tableaux d'objets préfixés par `...` et suivant le dernier paramètre formel de la fonction :

```
/// <summary>
/// journalise un événement (message paramétré {0}...)
/// </summary>
void log(String^ message, ... array<System::Object^> ^ p );
```

h. Les propriétés

Pas de plate-forme Microsoft sans propriété ni événement : ce sont des éléments indispensables à la programmation graphique tout d'abord mais bien plus, généralement, si l'on y regarde de plus près. Une propriété est un type particulier de membre qui s'apparente à un champ auquel on définit des accesseurs pour préciser son comportement :

- ˆ Lecture seule,
- ˆ Écriture seule,
- ˆ Formatage des valeurs en lecture ou en écriture,
- ˆ Contrôle de cohérence en lecture ou en écriture,
- ˆ Calcul d'une valeur en lecture ou en écriture...

Les applications des propriétés sont innombrables et la quasi-totalité de l'API .NET repose sur leur emploi plutôt que sur des champs publics.

```
ref class Utilisateur
{
protected:
    String^ login, ^ password;
    bool isLoggedIn;

public:
    Utilisateur()
    {
        isLoggedIn = false;
    }

    // une propriété en lecture / écriture
    property String^ Login
    {
        String^ get()
        {
            return login;
        }

        void set(String^ value)
        {
            login = value;
        }
    }

    // une propriété en écriture seule
    property String^ Password
    {
        void set(String^ value)
        {
            password = value;
        }
    }

    void Authenticate()
    {
        isLoggedIn = (login=="..." && password=="xxx");
    }
}
```

```

// une propriété en lecture seule
property bool IsLogged
{
    bool get()
    {
        return isLogged;
    }
}

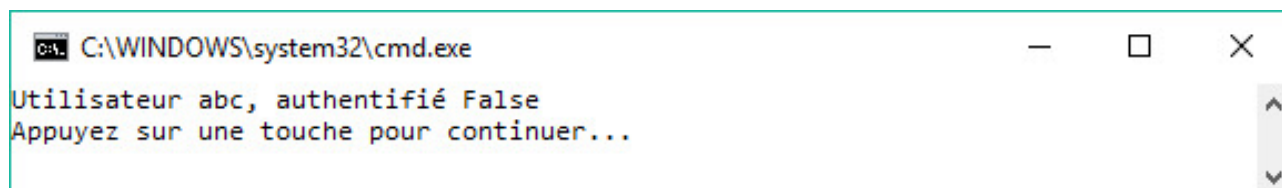
};

int main(array<System::String ^> ^args)
{
    Utilisateur^ user1 = gcnew Utilisateur();
    user1->Login = "abc";
    user1->Password = "098";
    user1->Authenticate();

    Console::WriteLine("Utilisateur {0}, authentifié {1}",
        user1->Login, user1->IsLogged);

    return 0;
}

```



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The output of the program is displayed as follows:

```

Utilisateur abc, authentifié False
Appuyez sur une touche pour continuer...

```

i. Les délégués et les événements

Les délégués de C++ CLI succèdent aux pointeurs de fonctions peu sûrs du langage C. Nous l'aurons compris, tous les éléments faiblement typés - et donc potentiellement dangereux - du monde classique ont été revisités en .NET. Les délégués de C++ CLI ont les mêmes rôles que les pointeurs de fonctions : ils servent à définir des fonctions génériques, des fonctions auxiliaires...

L'exemple ci-dessous montre comment utiliser un délégué pour définir une fonction

générique de comparaison entre deux nombres :

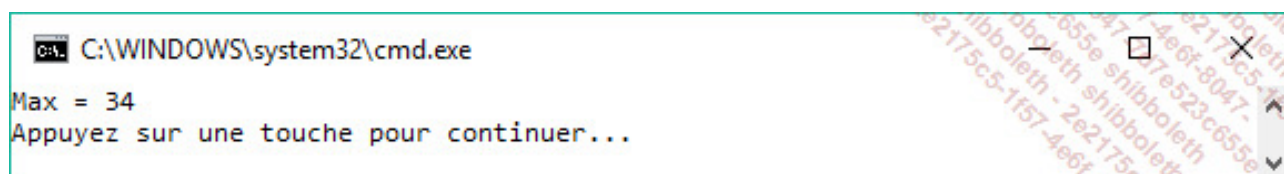
```
// un délégué est un type de fonction
delegate int del_compare(int a,int b);

// voici une fonction qui respecte la signature du délégué
int comparer_entier(int n1,int n2)
{
    return n1-n2;
}

// cette fonction emploie un délégué
int calcule_max(int a,int b,del_compare ^ f_comp)
{
    if(f_comp(a,b)>0)
        // appel de la fonction au travers du délégué
        return a;
    else
        return b;
}

int main(array<System::String ^> ^args)
{
    // déclaration et instanciation du délégué
    del_compare ^ pcomp;
    pcomp = gnew del_compare(&comparer_entier);

    // utilisation du délégué
    int max = calcule_max(34,22,pcomp);
    Console::WriteLine("Max = {0}",max);
    return 0;
}
```



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The output of the program is displayed as follows:

```
Max = 34
Appuyez sur une touche pour continuer...
```

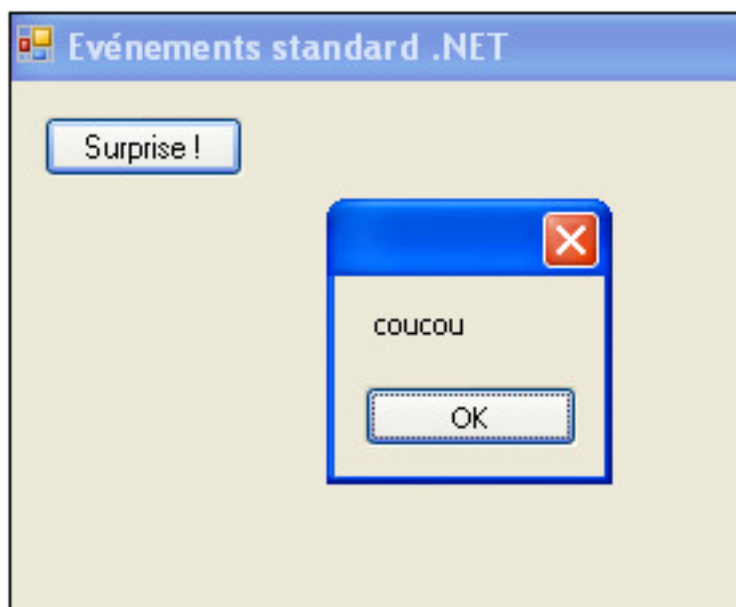
Le rôle des délégués ne se limite pas à l'algorithmie puisqu'ils sont également employés pour démarrer des traitements sur des threads séparés et à l'implémentation des

événements.

Un événement est un signal déclenché par un objet (ou une classe) qui peut être capté (on dit "géré") par une ou plusieurs méthodes. L'exemple habituel est celui de l'événement `Click` sur un objet `Button` :

```
this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
```

La méthode `button1_Click` est chargée d'afficher un message à l'écran :



Dans cet exemple, le délégué standard .NET est `System::EventHandler`. Mais nous verrons dans l'exemple du tableau comment déclarer ses propres signatures d'événements au moyen de délégués spécifiques.

j. Les méthodes virtuelles

Le fonctionnement habituel de C++, langage fortement typé avec phase d'édition des liens, est celui de méthodes non virtuelles en cas d'héritage de classes. Toutefois, c'est le principe inverse qui s'est imposé au fil des ans et de la montée en puissance des applications graphiques. Le programmeur C++ CLI doit donc être très assez précis quant à la mise en œuvre du polymorphisme. Le fait de déclarer une méthode comme étant

`virtual` exigera des surcharges au niveau des sous-classes, qu'elles précisent `override` ou `new`. Dans le premier cas, la méthode reste virtuelle et le polymorphisme est appliqué en déterminant dynamiquement le type de l'objet. Dans le second cas, le polymorphisme est rompu et c'est la déclaration de l'objet qui prime.

Voici un premier exemple utilisant `override` :

```
ref class Compte
{
protected:
    double solde;
    int num_compte;
public:
    Compte(int num_compte)
    {
        this->num_compte = num_compte;
        solde = 0;
    }

    virtual void Crediter(double montant)
    {
        if(montant>0)
            solde += montant;
        else
            throw gcnew Exception("Erreur");
    }

    void Afficher()
    {
        Console::WriteLine("Compte n°{0} Solde = {1}",num_compte,solde);
    }
};

ref class Livret : public Compte
{
protected:
    double taux;
public:
    Livret(int num_compte) : Compte(num_compte)
```

```

{
    taux = 0.0225;
}

virtual void Crediter(double montant) override
{
    if(montant>0)
        solde += montant*(1+taux);
    else
        throw gcnew Exception("Erreur");
}
};

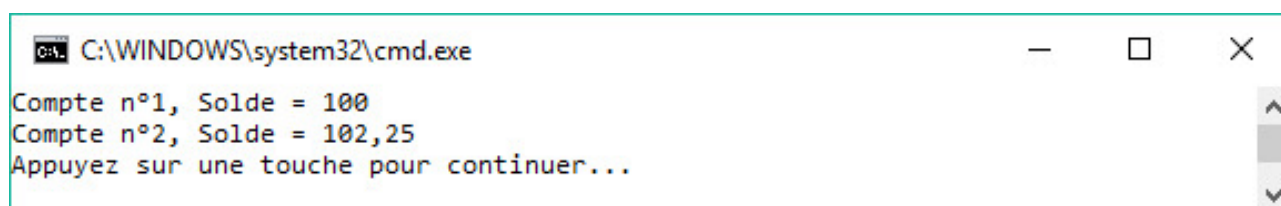
int main(array<System::String ^> ^args)
{
    Compte^ c1 = gcnew Compte(1);
    c1->Crediter(100);
    c1->Afficher();

    Livret^ l2 = gcnew Livret(2);
    Compte ^ c2 = l2;
    c2->Crediter(100);
    c2->Afficher();

    return 0;
}

```

Dans cet exemple, l'objet `c2` désigne en fait un `Livret`, donc le compte est crédité avec intérêts :



```

C:\WINDOWS\system32\cmd.exe
Compte n°1, Solde = 100
Compte n°2, Solde = 102,25
Appuyez sur une touche pour continuer...

```

En remplaçant `override` par `new`, la CLR aurait suivi la déclaration de `c2`, soit `Compte` et non `Livret`, et le compte n'aurait pas perçu d'intérêts.

k. Les classes abstraites et les interfaces

Par analogie avec les méthodes virtuelles pures de C++ classique, le langage C++ CLI propose les méthodes **abstraites**. Il s'agit de méthodes dont la définition n'a pas été donnée et qui doivent être implémentées dans des sous-classes concrètes avant d'être instanciables :

```
ref class Vehicule abstract
{
public:
    virtual void Avancer() abstract;
};

ref class Voiture : public Vehicule
{
public:
    virtual void Avancer() override
    {
        Console::WriteLine("Vroum");
    }
};
```

Nous mesurons la très grande proximité avec les méthodes virtuelles pures de C++, mais aussi le respect des mécanismes objets de .NET. Le mot-clé `abstract` est plus adapté que le pointeur NULL de C++, tandis que le mot-clé `override` souligne bien l'emploi du polymorphisme.

Le langage C++ CLI connaît aussi les interfaces, lesquelles s'apparentent à des classes totalement abstraites, mais sont surtout un moyen pour .NET de contourner l'héritage multiple au sein du framework.

Les interfaces de classes se définissent sans recourir aux superclasses puisqu'elles sont totalement abstraites ; c'est donc la forme `new` qui prévaut dans les classes d'implémentation :

```
interface class Icompte
{
    void Crediter();
};
```

```

    void Afficher();
};

ref class Compte : Icompte
{
public:
    virtual void Crediter() new
    {
        //...
    }

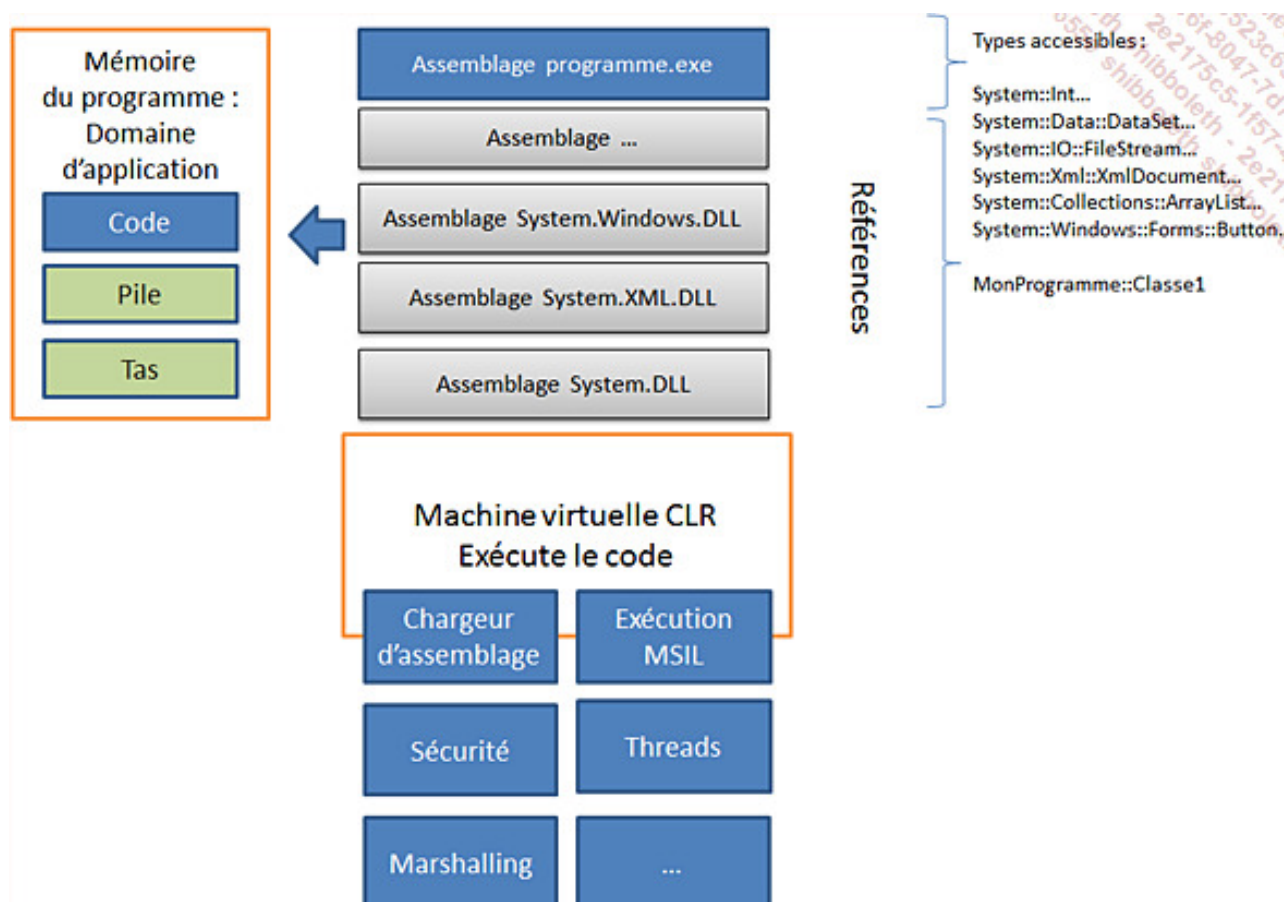
    virtual void Afficher() new
    {
        //...
    }
};

```

3. Le framework .NET

Microsoft propose un framework très complet pour supporter le développement de toutes ses applications. .NET Framework est l'implémentation de la CLR pour Windows accompagnée d'un immense réseau de classes organisées en espaces de noms et en assemblages. Tous les thèmes de la programmation sont abordés : réseau, graphisme, entrées-sorties, Internet, sécurité, XML, programmation système... Nous n'en donnerons qu'un aperçu pour aider le programmeur à démarrer avec cet environnement.

Un assemblage est une DLL ou un EXE qui contient des types - classes, énumérations, délégués, structures... - répertoriés dans des espaces de noms.

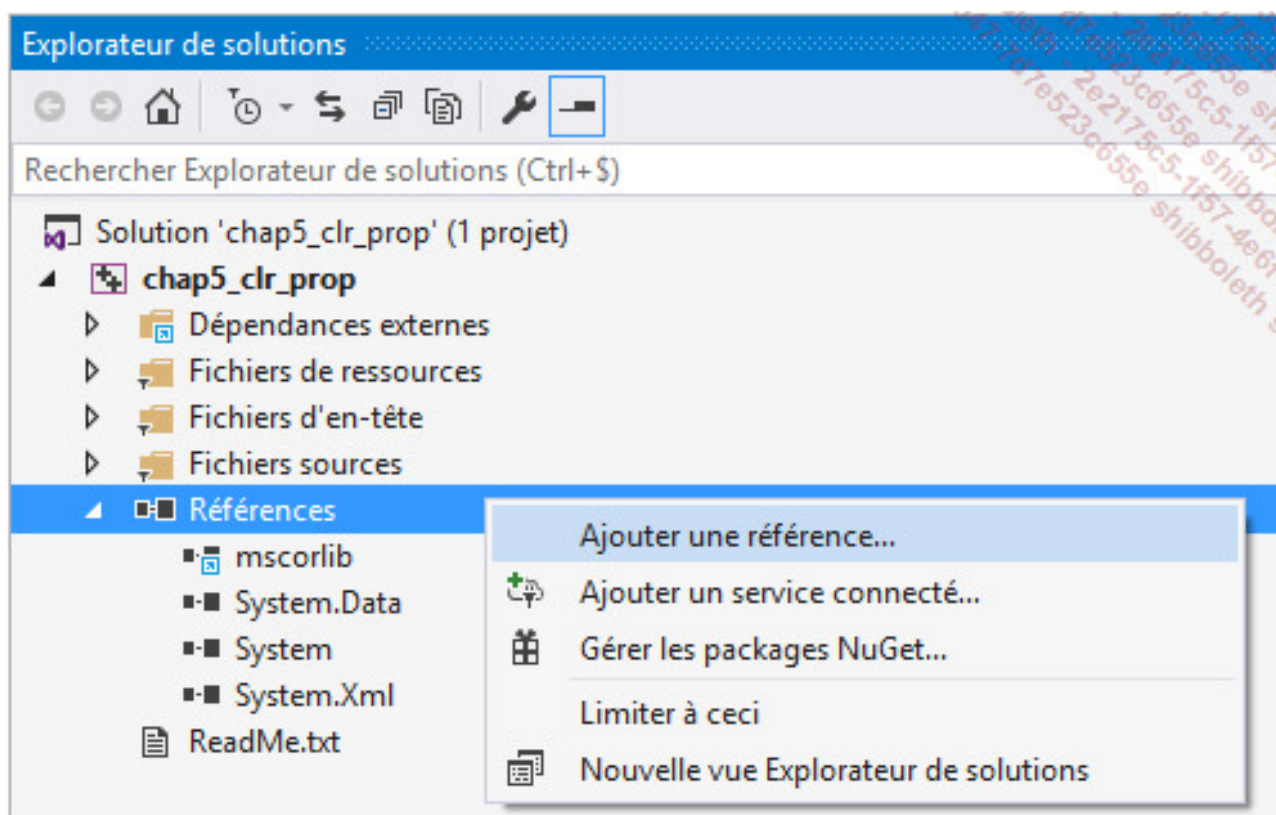


a. Les références d'assemblages

Pour accéder aux classes du framework, le programme doit d'abord référencer les assemblages qui les exposent, puis facultativement inclure les espaces de noms au moyen de l'instruction `using namespace`.

C'est depuis l'**Explorateur de solutions** que l'on modifie la liste des références d'assemblages.

Dans notre exemple ci-dessous, il y a déjà trois assemblages référencés :



b. L'espace de noms System::IO

Cet espace de noms rénove totalement l'ancienne façon d'accéder aux fichiers héritée du langage C. Il y a maintenant plusieurs types de classes spécialisées :

| | |
|---|--|
| <code>FileInfo</code> , <code>DirectoryInfo</code> | Opérations générales sur les fichiers et les répertoires - listes, déplacement... |
| <code>FileStream</code> | Flux fichier pour lire et écrire. |
| <code>Stream</code> | Classe abstraite pour lire ou écrire (fonctionne aussi avec le réseau). |
| <code>StreamReader</code> | Classe spécialisant un flux (<i>stream</i>) dans la lecture de caractères et de chaînes. |
| <code>StreamWriter</code> | Classe spécialisant un flux (<i>stream</i>) dans l'écriture de caractères et de chaînes. |
| <code>XmlTextReader</code> | Lecture continue d'un flux XML (aussi appelée API SAX). |
| <code>XmlTextWriter</code> | Écriture au format XML. |
| <code>StringReader</code> | Flux capable de lire dans une chaîne. |
| <code>StringWriter</code> | Flux capable d'écrire dans une chaîne. |
| <code>MemoryStream</code> | Flux mémoire. |

Voici un petit exemple d'écriture dans un fichier :

```
// utilise le contrôle de sélection de fichier
if(saveFileDialog1->ShowDialog() ==
    System::Windows::Forms::DialogResult::Cancel)
return;
```



```

// nom du fichier sélectionné
String^ filename = saveFileDialog1->FileName;

// ouverture d'un flux fichier en écriture
FileStream ^ fs = gcnew FileStream(filename, FileMode
    :: Create, FileAccess::Write);

// adaptation d'un flux texte sur le flux fichier
StreamWriter ^ sw = gcnew StreamWriter(fs);

// écriture d'une chaîne dans le flux
sw->Write("Ecriture dans un fichier");
sw->Flush(); // purge du flux texte bufferisé

// fermeture des flux
sw->Close();
fs->Close();

```

c. L'espace de noms System::Xml

Cet espace de noms propose de très nombreuses classes pour manipuler des données XML à l'aide des API SAX, DOM, XSL, XPath, XML schémas... La prise en charge de XML est donc très complète et fait figure de référence. L'exemple qui suit montre comment charger un extrait XML et l'analyser :

```

// un extrait XML
String ^ s ="<villes><ville>Paris</ville><ville>Nantes
    </ville><ville>Annecy</ville></villes>";

// instanciation d'un objet DOM
XmlDocument ^ doc = gcnew XmlDocument();

// initialisation de l'arbre DOM avec l'extrait XML
doc->LoadXml(s);

// parcours de l'arbre

```

```

XmlNode ^ n = doc->DocumentElement->FirstChild;
while( n != nullptr)
{
    String ^ v = n->InnerText;
    listBox1->Items->Add(v);

    // prochain noeud collatéral
    n = n->NextSibling;
}

```

d. L'espace de noms System::Data

Cet espace de noms est constitué des éléments d'accès déconnecté aux données relationnelles : DataSet, DataTable, DataView, DataRelation, DataRow, DataColumn... L'exemple suivant instancie une table `DataTable` et l'associe à la source de données d'un composant DataGridView :

```

// création d'une table et définition de 2 colonnes
DataTable ^ dt = gcnew DataTable();
dt->Columns->Add("Utilisateur");
dt->Columns->Add("Login");

// une ligne de données
DataRow ^ dr;

// la ligne est instanciée selon le modèle de la table
dr = dt->NewRow();
dr["Utilisateur"] = "Jean Valjean"; // 1ère valeur
dr["Login"] = "jval"; // 2ème valeur
dt->Rows->Add(dr); // rattachement à la table

// autre façon d'indexer les colonnes
dr = dt->NewRow();
dr[0] = "Cosette";
dr[1] = "cosy";
dt->Rows->Add(dr);

```

```
// encore une ligne
dr = dt->NewRow();
dr[0] = "Thénardier";
dr[1] = "bistr";
dt->Rows->Add(dr);

// "affichage"
dataGridView1->DataSource = dt;
```

e. L'espace de noms System::Collections

Les structures de données de l'espace de noms `System::Collections` ont fait partie de la première version de .NET Framework. Il s'agit de collections faiblement typées, qui exploitent pour beaucoup l'héritage de la classe `System::Object` par tous les types managés.

On retrouve dans cet espace de noms des piles, des files, des listes (tableaux dynamiques...). Voici justement un exemple d'emploi de la classe `ArrayList`. On découvre ici les avantages et les contraintes de l'approche faiblement typée. Il est très aisé d'attacher toutes sortes d'objets dans une liste, mais parfois plus délicat de retrouver leur type :

```
// un tableau liste contient des objets - valeurs ou références
ArrayList ^ tab = gcnew ArrayList();

tab->Add("Toulouse"); // ajout d'un objet référence
tab->Add((wchar_t)'P'); // ajout d'un objet valeur
tab->Add(123); // ajout d'un objet valeur

// crée 2 objets valeurs pour faciliter la détermination du type
wchar_t c = 0;
Object ^ ochar = c;

int i=0;
Object ^ oint = i;

// énumération de la collection
```

```

for each(Object ^ o in tab)
{
    String ^ v;
    v = o->ToString(); // version textuelle de l'objet

    // teste le type de l'objet
    if(dynamic_cast<String ^>(o) != nullptr)
        v = v + " est une String";

    // pour les types valeurs, dynamic_cast ne s'applique pas
    if(o->GetType()->Equals(ochar->GetType()))
        v = v + " est un char";

    if(o->GetType()->Equals(oInt->GetType()))
        v = v + " est un entier";

    // affichage
    listBox1->Items->Add(v);
}

```

Évidemment, si le type de la collection est homogène, il est plus simple d'utiliser systématiquement l'opérateur `dynamic_cast` voire l'opérateur de transtypage par coercition.

f. L'espace de noms `System::Collections::Generic`

Apparus avec la deuxième version de .NET Framework, les conteneurs génériques ont largement simplifié l'écriture des programmes puisqu'ils sont paramétrés - comme avec la STL - avec un type spécifique d'objet.

| | |
|----------------------------------|---|
| <code>Comparer<T></code> | Classe de base pour implémenter des algorithmes de tris génériques. |
| <code>Dictionary<T></code> | Table de hachage générique. |
| <code>LinkedList<T></code> | Liste générique doublement chaînée. |
| <code>List<T></code> | Liste générique. |
| <code>Queue<T></code> | File d'attente générique (aussi appelée pile FIFO). |
| <code>SortedList<T></code> | Liste générique dont les éléments peuvent être triés. |
| <code>Stack<T></code> | Pile générique (aussi appelée pile LIFO). |

L'exemple suit :

```
// cette liste ne contient que des String
List<String^> ^ liste = gcnew List<String^>();

// ajout d'objets d'après le type indiqué
liste->Add("Bordeaux");
liste->Add("Pau");
liste->Add("Laval");

// le parcours énumère les String
for each(String^ s in liste)
    listBox1->Items->Add(s);
}
```

g. Le portage de la STL pour le C++ CLI

Microsoft a tenté puis abandonné le projet de partage de la STL sous .NET.

4. Les relations avec les autres langages : C#

Notre démarche n'est pas ici de comparer C++ CLI et C#, mais plutôt d'illustrer les possibilités d'échanges et de partage entre les langages CLS (*Common Language Specification*).

Tous ces langages - C++ CLI, C#, VB.NET, F#... - partagent un socle commun, la CLR, qui exécute un code intermédiaire MSIL indépendant. Chaque programmeur peut ainsi choisir son langage sans se préoccuper des problèmes d'interopérabilité qui avaient largement nui au développement d'applications pendant tant d'années. Le choix est alors plus une question d'aptitude, de préférence, de quantité de code existant, ce qui est beaucoup plus positif que de procéder par élimination.

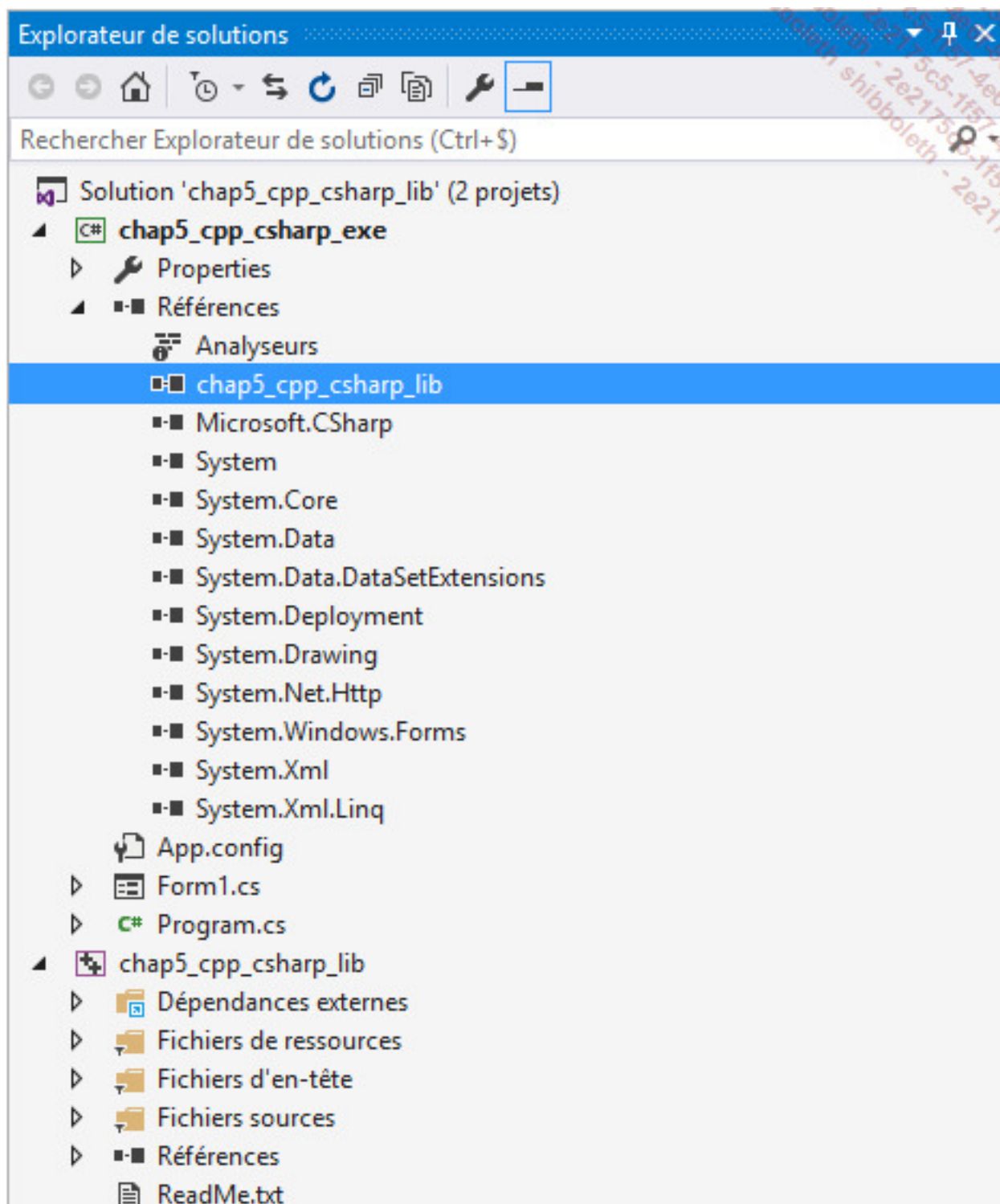
Nous proposons ici l'exemple d'une DLL écrite en C++ et d'une application graphique programmée en C#. Commençons par la DLL qui n'est constituée que d'une classe :

```
public ref class Utilitaire
{
public:
    static property DateTime Heure
    {
        DateTime get()
        {
            return DateTime::Now;
        }
    }

    static double cosinus(double a)
    {
        return 1 - a*a / 2;
    }
};
```

Après avoir créé un projet d'application console C#, nous ajoutons une référence à la DLL, ce qui est la condition pour que le programme C# puisse atteindre les types publics définis

précédemment :



Le système d'aide à la saisie IntelliSense de Visual Studio retrouve bien les types et les méthodes publics :

```
private void Form1_Load(object sender, EventArgs e)
{
    lbl_message.Text = chap5_cpp_csharp_lib.Utilitaire.
}

```

cosinus
Equals
Heure
ReferenceEquals

DateTime chap5_cpp_csharp_lib.Utilitaire.Heure { get; }

Il n'y a plus qu'à tester :

