C++/CLI & CX の本

haseham

はじめに

「**C++/CLI**」は、プログラミング言語「**C++**」の拡張言語です。

　わかりやすくいうと、「**C++/CLI**」のソース上では、
従来の**C++**のソースに加えて、「**.Net Framework**」のクラスライブラリを用いたソースを
混ぜて書くことができるのです。

　また、**.Net**の実行環境である「**CLR**」には、ガベージコレクションが備わっていますから、
.Net側のクラスインスタンスは、自動的に解放されます。

　実行速度の面では、**C#**で書いたとしても、**C++/CLI**で書いたとしても、
結局、中間表現である「**MSIL**」に変換されるため、そう変わりはないはずですが、
C++/CLIでは、部分的に、**C++**のネイティブコードで書くことができるため、
その点では有利なのではないかと考えられます。

　プログラムの勉強をはじめたい方や、始めて間もない方は、まず、
拙作の「ハム太と作る**C/C++**ライブラリ**0** 基本構文編」をご一読下さい。

　一度、マスターしてしまえば、**C#**の簡単さと、**C++**のパフォーマンスを
両方とも手に入れることができますので、頑張ってトライしてみてください。

.Net Framework

・.Netは、Javaとよく似ていて、
ガベージコレクションが付いていて、
実行環境と便利なクラスライブラリが
セットになったものです。

・さいきんのWindowsには、必ずこれの最新版が入っていますから、
プログラムを動かすために、何かをダウンロードする必要はありません。

・ただ、開発する場合は、「Visual Studio」と、
「Windows SDK」の最新版が必要です。

→「Windows SDK」のインストールはこちら。
→「Visual Studio 2015」(無償版) のインストールはこちら。

・コードエディタも配布されています。↓

→「Visual Studio Code」のインストールはこちら。

・このエディタは、Mac用や、Linux用のものも用意されています。

・.Netが、Javaと大きく異なる点としては、
Javaだと、プログラムをコンパイルする際に、
バイトコードに変換しておいて、
それを実行時に、仮想マシンというプログラムに読み込んで動かすのですが、
.Netの場合は、プログラムが実行される直前に、コンパイルが行われ、
ネイティブコードに変換されてから実行されます。

・もう一つ、対応するプラットフォームについてですが、
以前は、.Netといえば、Windowsでしか動かなかったんですが、
現在では、他のプラットフォームでも動作するバージョンが
オープンソースで開発されています。↓

https://msdn.microsoft.com/ja-jp/library/dn878908(v=vs.110).aspx

対応するバージョン

.NETのバージョン	CLRのバージョン	おもな新要素	含まれているVisual Studioのバージョン	インストール可能なWindowsのバージョン	インストール可能なWindows Serverのバージョン
2.0	2.0	・ジェネリック ・ASP.NET2.0	2005	XP	2003～2008 SP2
3.0	2.0	・WPF・WCF・WF・WCF	–	Vista	2003～2008 R2
3.5	2.0	・LINQ	2008	Vista以降	2003～2012 R2
4	4	・ポータブルクラスライブラリ	2010	Vista～7	2003～2008 R2
4.5	4	・ストアアプリ	2012	Vista～8	2008 SP2～2012
4.6	4	・.NET ネイティブによるコンパイル	2015	Vista～10	2008 SP2～2012 R2
4.6.1	4			7～10の最新版	2008 SP2～2012 R2

・詳細については、msdnのこちらのページで確認してください。↓
→ .NET Framework のバージョンおよび依存関係

・各APIのリファレンスは、こちらに一覧があります。↓
→ Microsoft API とリファレンスのカタログ

ハローワールド

・基本的には、**.Net**のクラスライブラリを使います。↓

```
#using <mscorlib.dll>
using namespace System;

int main()
{
    Console::WriteLine( L"Hello, World!" );

    return 0;
}
```

・**.Net**のクラスライブラリを利用する場合は、インクルードは不要ですが、
上記のように、利用するクラスの名前空間を**using namespace**指定しておけば、
名前空間を省略して書くことができます。

・ちなみに、上記の「**WriteLine**」メソッドは、「**Console**」クラスの静的メンバで、
「**System**」名前空間に属しています。

・**C++/CLI**では、さらに、**C**や、**C++**のソースを混ぜて書くこともできます。↓

```
#using <mscorlib.dll>
using namespace System;

#include <stdio.h>
#include <iostream>
using namespace std;

int main()
{
```

```
    printf( "Hello, World!" );
    cout << "Hello, World!" << endl;
    Console::WriteLine( L"Hello, World!" );

    return 0;
}
```

・ちなみに、「**MSCOREE.dll**を開けませんでした。」というエラーが出た場合は、
このライブラリファイルへのパスを通す必要があります。

・「**Visual Studio**」のメニューバーから、[プロジェクト]をクリックし、
開いた画面で、左側のツリービューの[リンカ]をクリックし、次に[全般]をクリックします。

・右側の画面に、[追加のライブラリパス]という入力欄がありますから、
そこに、このライブラリファイルへのフルパスを入力します。

・パスがわからない場合は、**Windows**の検索機能で調べてください。

・通常は、「**C:¥Program Files (x86)¥Windows Kits¥**」以下あたりの
どこかにあります。

値型

・**C/C++**の基本データ型は、
　コンパイル時に、**.net**のプリミティブ型に変換されます。

C++基本ﾃﾞｰﾀ型	C++/CLI.Net のﾌﾟﾘﾐﾃｨﾌﾞ型
bool	Boolean
char	SByte か Byte (※注1)
signed char	SByte
unsigned char	Byte
short	Int16
unsigned short	UInt16
int	Int32
unsigned int	UInt32
long	Int32　(※注2)
unsigned long	UInt32 (※注2)
long long int	Int64
unsigned long long int	UInt64
float	Single
double	Double
long double	Double (※注2)
wchar_t	Char

※注1 ... IsSignUnspecifiedByte による。
※注2 ... IsLong による。

文字列

・文字列は、System::Stringクラスで管理します。 ↓

```
String^ p_text1 = L"あいうえお";
```

・**String**クラスは、内部に、**Unicode**文字列を持っています。
(ANSI文字列 (char型) を代入した場合でも変換されます。)

・文字列リテラルの手前に付ける「**L**」は、省略できます。

・すべての文字列は、**String**クラスのインスタンスに変換されます。

・また、文字列同士は、「**+**」演算子で結合できます。 ↓

```
String^ p_text1 = "あいうえお" + "かきくけこ";
```

・この場合は、内部的には、**Concat**メソッドと同じ処理が行われています。

・ちなみに、**String**クラスのハンドルに、文字列を代入した時にも、
新しいインスタンスが作成され、代入されています。

参照クラスと値クラス

・「**C++/CLI**」では、ヒープメモリ領域上に、
「**CLR**」(.NETの実行環境)が管理する「マネージヒープ」という領域があり、
.**NET**系のクラスのインスタンスは、ここに置かれます。

・そして、それらのインスタンスは、空き領域が少なくなった時点で、
並べ替えられたり、参照されていないものから解放されたりします。

・従来のポインタや参照では、こうした並べ替えに対応できないため、
「追跡ハンドル」という、更新機能のついたポインタが導入されました。

・マネージヒープに置かれるインスタンスは、
すべて「マネージクラス」として宣言する必要があります。

・マネージクラスには、「参照クラス」と、「値クラス」があります。

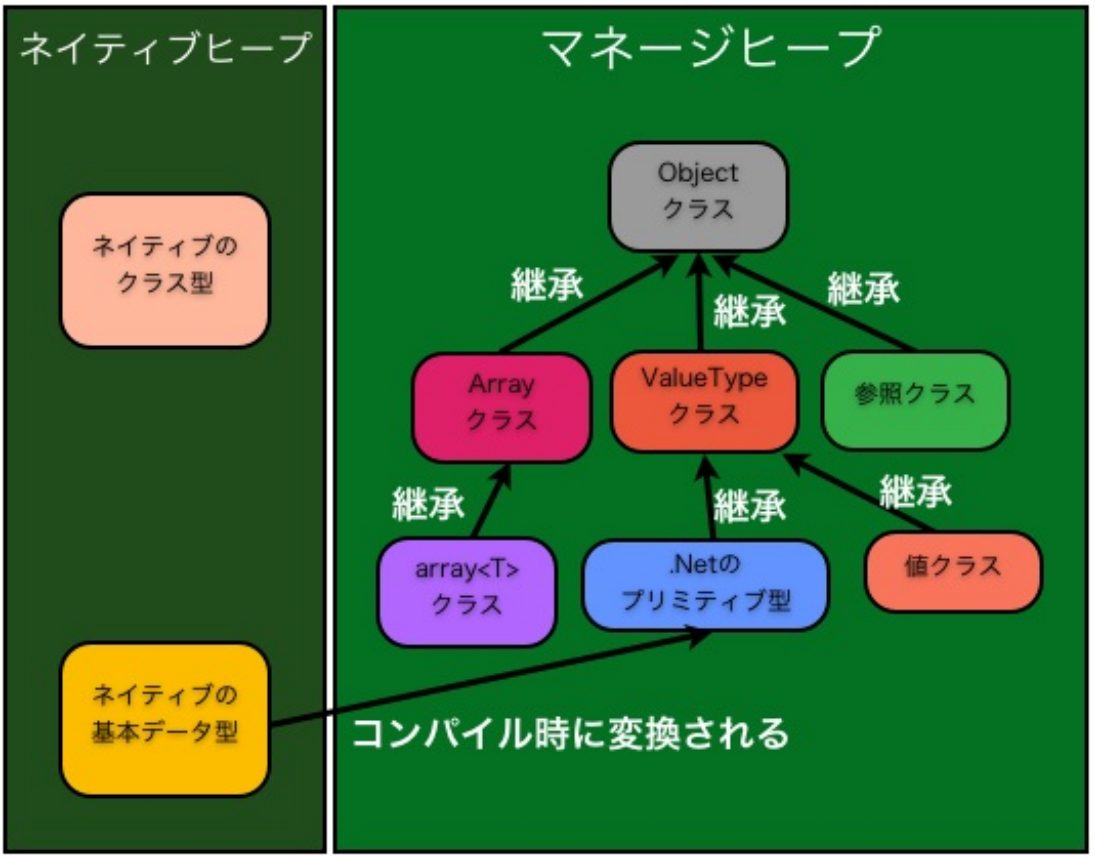
・参照クラスのインスタンス化は、基本的に、**gcnew**演算子で行い、
そのインスタンスは、マネージヒープ領域上に置かれます。

・一方、「値クラス」のインスタンスは、ローカル変数としてインスタンス化され、
そのインスタンスは、スタック領域上に置かれます。

・参照クラスは、暗黙的に、**System::Object**クラスの派生クラスとなります。

・値クラスは、暗黙的に、**System::ValueType**クラスの派生クラスとなります。

・ちなみに、プリミティブ型も値クラスです。



・一見してわかるように、ネイティブ側とマネージ側では、
クラスの系統がまったく異なるため、クラスごとのやりとりはできません。

・しかし、基本データ型の値には、互換性があるため、
単一値をやりとりすることは可能です。

・**C++/CLI**では、原則的に、マネージクラスを宣言し、利用します。

・参照クラスのインスタンスは、**CLR**によって管理されているため、
アプリケーションプログラムの側で**delete**する必要はありません。

・また、**.Net**のクラスライブラリのクラスも、参照クラスですから、
同じようにして扱うことができ、データのやり取りもスムーズに行えます。

追跡ハンドル

・マネージクラスへのポインタは、追跡ハンドルといい、次のように記述します。↓

```
Class1^ p_obj1 =gcnew Class1();
p_obj1->Data1 = 120;
*p_obj2 = *p_obj1;
```

追跡参照

・マネージクラスへの参照は、追跡参照といい、次のように記述します。↓

```
 ValueClass1 obj1;
ValueClass1% r_obj1 = obj1;
r_obj1.Data1 = 120;
```

ボックス化

・値クラスは、原則的に、ローカル変数用のクラスですが、
　次のようにすると、値を、マネージヒープにコピーすることができます。↓

```
int      v1 = 120; // C/C++のデータ型の変数「v1」を宣言する。
Int32^ p1 = v1; // 参照クラスのインスタンスが作られ、
                  //  そのアドレスが、追跡ハンドル「p1」に代入される。
int      v2 = p1; // メンバとして保持されていた値が取り出され、代入される。
```

・ローカル変数「v1」の持つ値は、スタック領域上に置かれていますが、
　追跡ハンドル「p1」がポイントしている参照クラスのインスタンスは、
　マネージヒープ領域上にあり、代入された値は、その中に、コピーされています。

・このように、スタックからマネージヒープへと、値がコピーされることを
　「ボックス化」といい、その逆を、「ボックス化の解除」といいます。

マネージ配列 **array<T>**

・マネージ配列を確保する場合は、次のように書きます。↓

```
array<Class1>^ p_array1 = gcnew array<Class1>( 3 );
```

・この配列の要素は、参照クラス「**Class1**」のインスタンスで、要素数は、**3**です。

・さらに、要素の初期化も行いたい場合は、次のように書きます。↓

```
array<Class1>^ p_array1 = gcnew array<Class1>( 3 ){ 0, 1, 2 };
```

・マネージ配列は、System::Arrayクラスのインスタンスですから、
CopyToメソッドや、Lengthプロパティなどを利用することができます。

・マネージ配列では、要素に、**C/C++**の基本データ型の値を格納することもできます。↓

```
array<int>^ p_array1 = gcnew array<int>(3){0,1,2};
interior_ptr<int> p_item1 = &p_array1[1]; // アドレスは、内部ポインタで保持する。
Int32 v1 = p_item1; // 1が代入される。
```

・要素は、もちろん、**Int32**クラスのインスタンスに変換されていて、

要素の値は、その中で、データメンバとして保持されています。

・アドレスは、内部ポインタで持ちます。

多次元のマネージ配列

・多次元のマネージ配列を、ローカル変数として宣言する場合は、次のように書きます。↓

```
// C/C++ だと、 int     array1[2][2];
// C#    だと、 Int32 array1[ 2, 2 ];
array<Int32,2>^ p_array1;
array<Int32,2>^ p_array2 = { {0,1}, {0,1} };
```

・動的に確保する場合は、次のように書きます。↓

```
// C/C++ だと、 int *     p_array1 = new int[ 2 * 2 ];
// C#    だと、 Int32[,] p_array1 = new Int32[ 2, 2 ];
array<Int32,2>^ p_array1 = gcnew array<Int32,2>( 2, 2 );
array<Int32,2>^ p_array2 = gcnew array<Int32,2>( 2, 2 ) { {0,1}, {0,1} };
```

・要素に、値を代入する場合は、次のように書きます。↓

```
// C/C++ だと、p_array1[0][0] = 120;
// C#    だと、 p_array1[2,2]  = 120;
p_array1[0,0] = 120;
```

・**C/C++**と同じ添え字を使いたい場合は、次のようにして配列を確保します。↓

```
array< array<int>^ >^ p_array1 = gcnew array< array<int>^ >( 2 );
p_array1[0][0] = 120;
```

---

for each 文

・まず、マネージ配列の要素値を列挙してみましょう。↓

```
using namespace System;

void main()
{
   array<Int32>^ p_a1 = gcnew array<Int32>{ 0, 1, 2 };

   for each ( Int32 item in p_a1 )
   {
      Console::WriteLine( item );
   }
}
```

・次に、**.Net**のコレクションクラス。(非ジェネリック)　↓

```
using namespace System;
using namespace System::Collections;

void main()
{
   ArrayList^ p_a1 = gcnew ArrayList( gcnew array<Int32>{ 0, 1, 2 } );

   for each ( Int32 item in p_a1 )
   {
      Console::WriteLine( item );
   }
}
```

・最後に、ジェネリックのコレクションクラス。↓

```
using namespace System;
using namespace System::Collections::Generic;

void main()
{
   List<Int32>^ p_a1 = gcnew List<Int32>( gcnew array<Int32>{ 0, 1, 2 } );

   for each ( Int32 item in p_a1 )
   {
      Console::WriteLine( item );
   }
}
```

・**for each**文のところは、いずれの場合でも、まったく同じです。

参照クラスの宣言 **ref class**

```
ref class Class1
{

public:

   // デフォルトコンストラクタ（これが無いとインスタンス化できない。）
   Class1();

   // コピーコンストラクタ（インスタンスを代入したり、配列を宣言する場合に必要。）
   Class1( const Class1% src_ );

   // デストラクタ（delete演算子で解放する場合に必要。）
   ~Class1();

protected:

   // ファイナライザ（ガベコレによる解放時に呼ばれる。）
   !Class1();
};
```

・参照クラスには、次の制約があります。↓

・ｱﾝﾏﾈｰｼﾞｸﾗｽを継承できない。
・多重継承ができない。
・friend を使えない。
・volatileを使えない。(ﾛｰｶﾙ変数には使える。)
・constﾒｿｯﾄﾞを作れない。
・sizeof演算子が使えない。(型名でもｲﾝｽﾀﾝｽでも無理。)

・new や delete をｵｰﾊﾞｰﾛｰﾄﾞできない。

デストラクタとファイナライザ

・ファイルハンドルや、DBの接続ハンドルなどのシステムリソースは、
使い終わったら、すぐに解放しなければいけません。

・しかし、ガベージコレクションによる解放は、いつ起きるのかわかりません。

・したがって、システムリーソスへのハンドルをデータメンバとして持つクラスでは、
独自にデストラクタを実装し、その中で、システムリソースの解放を行う必要があります。

・ ちなみに、C#では、こうしたクラスでは、IDisposableインターフェイスを実装し、
Disposeメソッドの中で、リソースの解放処理を行うことになっています。

(ちなみに、CLIのデストラクタは、コンパイル時に、Disposeメソッドに変換される。)

・ハンドルにnullptrを代入しても、インスタンスは即座には解放されません。
(マネージヒープ領域に、空きがなくなった時点で、解放される。)

・ファイナライザを実装していると、ガベージコレクションによって
インスタンスが解放された時点で、ファイナライザが呼ばれます。

・アプリの終了時まで、ガベージコレクションが動作しなかった場合は、
アプリの終了時に、ファイナライザが呼ばれます。

・しかし、delete演算子でインスタンスを解放すると、デストラクタが呼ばれ、
ファイナライザは呼ばれません。(ここ重要)

・こうした事情があることから、delete演算子を使って
インスタンスを解放する可能性がある場合は、通常は、
デストラクタの中でファイナライザを呼ぶようにしておき、
ファイナライザの中で、データメンバの解放処理を行うようにします。

静的コンストラクタ

・**C++/CLI**では、「静的コンストラクタ」を宣言することができます。↓

```
static RefClass1()
{
    member1 = 100;
}
```

・これは、クラスの静的データメンバを、まとめて初期化するためのもので、
静的データメンバのいずれかが、最初にアクセスされた時点で呼ばれます。

・ちなみに、**C++/CLI**では、データメンバは、宣言時に初期化ができます。↓

```
ref class RefClass1
{
    static int member1 = 100; // 静的ﾃﾞｰﾀﾒﾝﾊﾞの宣言。
};
```

・それから、**C++/CLI**では、データメンバの値は、
自動的に **0** で初期化されています。

リテラル literal initonly

・次の２つは、同じ意味を持ちます。↓

```
static const Int32 member1 = 100;
```

```
literal Int32 member1 = 100;
```

・**C++/CLI**では、**static const** は、定数とはみなされません。

・**static const** とする箇所では、なるべく **literal** を使います。

・また、初期化だけを許可したい場合は、**initonly** を使います。↓

```
initonly Int32 member3;

static RefClass1() // static ｺﾝｽﾄﾗｸﾀ
{
    member3 = 100; // 初期化できる。
}
```

・**initonly** は、おもに、値クラスのデータメンバでよく使います。

・**const** が、コンパイル時に解決される (値が決定される) のに対して、
initonly は、プログラムの実行時に解決されます。

・このため、設定ファイルなどから、初期設定値を読み込んで
初期化することができます。

クラスのアクセス指定子

・**.Net**で開発したライブラリに含まれているクラスの型情報は、
原則的には、すべてライブラリの外にも公開されていますが、
中には、公開したくないクラスもあると思います。

・そんな場合には、次のようにして、クラスを宣言します。↓

```
private ref class RefClass1
{

};
```

・クラス宣言の最初にあるアクセス指定子が「**private**」になっています。

・クラスのアクセス指定子には、その他にも次のようなものがあります。↓

ｱｸｾｽ指定子	意味
public	ｸﾗｽ外のｺｰﾄﾞから利用できる。
protected	派生ｸﾗｽでなら利用できる。
private	ｸﾗｽ内でしか利用できない。
internal	同じｱｾﾝﾌﾞﾘ内でなら利用できる。
public protected protected public	同じｱｾﾝﾌﾞﾘ内なら利用できる。 また、別のｱｾﾝﾌﾞﾘでも、派生ｸﾗｽでなら利用できる。
protected private private protected	同じｱｾﾝﾌﾞﾘ内の派生ｸﾗｽでなら利用できる。

・「アセンブリ」というのは、モジュールを１つにまとめたもので、
個々のライブラリと考えるとわかりやすいかと思います。

・ちなみに、省略した場合は、「**public**」であるとみなされます。

メソッドのオーバーライド **override**

・**C++/CLI**では、メソッドをオーバライド(上書き)する場合には
次のように書きます。↓

```
// 親クラスの宣言
ref class ParentClass1
{

public:

    virtual void Method1()
    {
    }

};
```

```
// 子クラスの宣言
ref class ChildClass1 : ParentClass1
{
    virtual void Method1() override
    {
    }
};
```

・派生クラスでの上書きを禁止する場合は、**sealed** を付けます。↓

```
virtual void Method1() override sealed
{
}
```

抽象クラス **abstract**

・「抽象クラス」は、純粋仮想関数をメンバに持つクラスです。

・抽象クラスの宣言は、次のように書きます。↓

```
ref class Class1 abstract
{
public:
   // このメソッドが、純粋仮想関数。↓
   virtual void Method1() = 0;
};
```

・抽象クラスでは、メソッドの実装は一切行いません。

値クラスの宣言 **value class**

・値クラスは、次のようにして宣言します。↓

```
value class ValueClass1
{

};
```

・値クラスのメリットは、インスタンスがスタック領域に置かれるため、

生成時のオーバーヘッドが少なく、それ以降も、ガベージコレクションの
負担を軽減できるということです。

・しかし、値クラスには、次の制約があります。↓

・ クラス継承ができない。
・コンストラクタ、デストラクタ、ファイナライザは持てない。(メソッドは可)
・代入演算子のオーバーロードができない。
・データメンバは、原則的に、値クラスであるべきである。
(参照クラスや、システムリソースの管理には向いていない。)

・ちなみに、構造体は、次のようにして宣言します。↓

```
value struct ValueStruct1
{

};
```

・構造体のメンバは、暗黙的に **public:** であり、すべて公開されています。
・制約などはすべて、クラスの場合とまったく同じです。

・値クラスの性質からすると、構造体の方が向いているように思います。

クラス3系統の機能一覧 (早見表)

・ネイティブクラス、値クラス、参照クラスのそれぞれで
　メンバとして持つことができるものをまとめてみました。

	ﾈｲﾃｨﾌﾞｸﾗｽ	値ｸﾗｽ	参照ｸﾗｽ
変数	○	○	○
ﾎﾟｲﾝﾀ	○	○	○
ｸﾗｽのｲﾝｽﾀﾝｽ	○	×	×
値ｸﾗｽのｲﾝｽﾀﾝｽ	○	○	○
値ｸﾗｽのﾊﾝﾄﾞﾙ	×	○	○
参照ｸﾗｽのｲﾝｽﾀﾝｽ	×	×	○
参照ｸﾗｽのﾊﾝﾄﾞﾙ	×	○	○
ｺﾝｽﾄﾗｸﾀ	○	×	○
ﾃﾞｽﾄﾗｸﾀ	○	×	○
ﾌｧｲﾅﾗｲｻﾞ	×	×	○
ﾒｿｯﾄﾞ	○	○	○
継承	○	×	○
ﾛｰｶﾙ変数宣言	○	○	○
new	○	○	×
gcnew	×	○	○

・メンバ以外については、値クラスは継承ができません。

・次に、参照クラスは、ローカル変数のように宣言することもできますが、
　その場合でも、**gcnew**演算子によってインスタンス化され、
　インスタンスは、マネージヒープ上に置かれます。

インターフェイスの宣言 **interface class**

・参照クラスでは、多重継承が禁止されています。
・その代わりに、インターフェイスを利用します。

・インターフェイスの宣言は、次のように書きます。↓

```
interface class Interface1
{
    void Method1(); // プロトタイプ宣言のみ。
};
```

・インターフェイスの実装は、次のように書きます。↓

```
ref class RefClass1 : public Interface1
{
   void Method1(){  } // 必ず実装する。
};
```

・インターフェイスには、次の制約があります。↓

・データメンバ、コンストラクタ、ファイナライザを
メンバとして持つことができない。
(逆にいうと、メソッド、デストラクタ、プロパティ、デリゲートは可。)

・メソッドは、暗黙的に、**public:** で **abstruct** となる。

・インターフェイスを実装するクラスは、そのインターフェイスが持つ
すべてのメソッドを、必ず実装する必要がある。
(ただし、その派生クラスでは、必ずしも実装する必要はない。)

・宣言の中では、**friend**が禁止されている。

列挙クラスの宣言 **enum class**

・**C++/CLI** では、列挙体の代わりに、「列挙クラス」を使用します。

・列挙クラスの宣言は、次のように書きます。↓

```
enum class Shingo
{
    Aka,
    Ki,
    Midori
};
```

・列挙クラスは、暗黙的に、**System::Enum**クラスの派生クラスとなります。

・各列挙子の定数値を取り出すには、静的キャストが必要です。↓

```
Singo singo1;
singo1 = static_cast<Singo>( 1 );
int num1 = static_cast<int>( singo1 );
```

・列挙子の名称を文字列として取得することもできます。↓

```
String^ p_name1 = singo1.ToString();
```

・指定した値を持っているか、判定することもできます。↓

```
bool is_defined = Enum::IsDefined( Singo::typeid, singo1 );
```

・すべての列挙子を、配列に格納することもできます。↓

```
array<Shingo>^ p_array1
  = static_cast<array<Shingo>^>( Enum::GetValues( Shingo::typeid ) );

for each ( Shingo cur in p_array1 )
{
    Console::WriteLine( cur );
}
```

プロパティの宣言 **property**

・**C/C++**では、データメンバの値を、取得したり、変更する際には、
アクセッサを使っていましたが、あまり直感的ではありませんでした。

・**CLI**では「プロパティ」を使うことができ、
代入演算子による値の取得・変更ができます。↓

```
p_obj1->Property1 = 100; // Setter

int value1 = p_obj1->Property1; // Getter
```

・プロパティの宣言は、次のように書きます。↓

```
property int Property1
{

public:
    int get()
    {
        return member1;
    }

private:
    void set( int value_ )
    {
        member1 = value_;
    }
}
```

・さらに、**get set** それぞれに、ｱｸｾｽ指定子を付けることができます。

デリゲートの宣言 **delegate**

・「デリゲート」は、関数ポインタを包んだようなクラスで、
　インスタンス化する際に渡した関数を、
　Invokeメソッドで呼び出すことができます。

・それでは、デリゲートの定義と呼び出し例を見ていきましょう。↓

```
using namespace System;

delegate void Delegate1(); // ﾃﾞﾘｹﾞｰﾄの定義。

// 関数の実装。 ( ※ ﾃﾞﾘｹﾞｰﾄと、ｼｸﾞﾈﾁｬが同じである必要がある。)
void Func1()
{
    Console::WriteLine( L"呼ばれて飛び出て以下省略" );
}

void main()
{
  Delegate1^ p_proc1 = gcnew Delegate1( Func1 ); // ｲﾝｽﾀﾝｽ化する。

  p_proc1->Invoke(); // 関数を呼び出す。
}
```

非同期デリゲートの宣言

・「非同期デリゲート」は、セットされた関数の処理を、
　別のスレッドで並列実行します。

・さらに、終了時の処理を行うコールバック関数を
　セットしておくこともできます。

・それでは、その宣言と使用例を見ていきましょう。↓

```
// 非同期ﾃﾞﾘｹﾞｰﾄの宣言。

delegate int AsyncDelegate1( int, int );
```

```
// 非同期ﾃﾞﾘｹﾞｰﾄで、開始時に呼ばれる関数。

int AsyncFunc1(  int left_, int right_ )
{
    return left_ + right_;
}
```

```
// 非同期ﾃﾞﾘｹﾞｰﾄの、終了時に呼ばれるｺｰﾙﾊﾞｯｸ関数。

using namespace System::Runtime::Remoting::Messaging; // AsyncResult

static void CallbackFunc1( IAsyncResult^ p_async_result_ )
{
  // 非同期ﾃﾞﾘｹﾞｰﾄの、結果を取り出すための非同期結果を、静的ｷｬｽﾄする。
  AsyncResult^ p_async_result =
                        static_cast<AsyncResult^>( p_async_result_ );

   // その非同期結果から、非同期ﾃﾞﾘｹﾞｰﾄを取り出す。
   AsyncDelegate1^ p_async_proc
```

```
  = static_cast<AsyncDelegate1^>( p_async_result->AsyncDelegate );

  // 非同期ﾃﾞﾘｹﾞｰﾄを完了させ、その結果を受け取る。( 非同期結果を渡している。)
  int result = p_async_proc->EndInvoke( p_async_result_ );

  // 結果を出力する。
  Console::WriteLine( "結果: " + result );
}
```

```
void main()
{
// ------------------------
// 非同期ﾃﾞﾘｹﾞｰﾄを作成する。

AsyncDelegate1^ p_ad1 = gcnew AsyncDelegate1( AsyncFunc1 );

// ------------------------
// その終了時に呼ばれるｺｰﾙﾊﾞｯｸ関数のﾃﾞﾘｹﾞｰﾄも作成しておく。
// ( そしてこれを、 BeginInvokeﾒｿｯﾄﾞの呼び出し時に渡す。)

AsyncCallback^ p_ac1 = gcnew AsyncCallback( CallbackFunc1 );

// ------------------------
// 非同期呼び出しを開始する。( これは別のｽﾚｯﾄﾞで動作する。)

p_ad1->BeginInvoke( 1, 2, p_ac1, gcnew array<Object^>(2){1,2} );

// ------------------------
// 非同期ﾃﾞﾘｹﾞｰﾄが、別のｽﾚｯﾄﾞで動作する中、
// ﾒｲﾝｽﾚｯﾄﾞでは、この無意味なﾙｰﾌﾟを実行し続ける。

for ( int i = 0; i < 10; i++ )
{
Console::WriteLine( "ｽﾘｰﾌﾟ中" );
System::Threading::Thread::Sleep(20); // ｽﾘｰﾌﾟ中。
}
```

```
// ----------------------
}
```

イベントの宣言 event

```
delegate void EventDelegate1(); // ｲﾍﾞﾝﾄ用ﾃﾞﾘｹﾞｰﾄの宣言。

// ﾎﾞﾀﾝｸﾗｽの宣言。
ref class Button
{

public:

   event EventDelegate1^ OnClick; // ｸﾘｯｸ時ｲﾍﾞﾝﾄの宣言。

   // ｸﾘｯｸｲ時のﾍﾞﾝﾄを実行するﾒｿｯﾄﾞ。
   void Click()
   {
      this->OnClick();
   }
};

void ClickEventHandler1(){} // 独自のｲﾍﾞﾝﾄﾊﾝﾄﾞﾗの実装。

void main()
{
   Button button1; // ﾎﾞﾀﾝ1の宣言。

   // 独自のｲﾍﾞﾝﾄﾊﾝﾄﾞﾗを、ｲﾍﾞﾝﾄ用ﾃﾞﾘｹﾞｰﾄに追加する。
   button1.Onclick += gcnew EventDelegate1( ClickEventHandler1 );

   button1.Click(); // ﾌﾟﾛｸﾞﾗﾑからｸﾘｯｸ時のｲﾍﾞﾝﾄを実行する。
}
```

内部ポインタ **interior_ptr<T>**

・追跡ハンドルは、ポインタと似ていますが、
　ポインタ演算や、アドレス値の比較ができません。

・いっぽう、ポインタには、追跡ハンドルのような
　ポイント先の更新機能が付いていません。

・「内部ポインタ」というのは、この両方の機能を兼ね備えたもので、
　たとえば、ある参照クラスがデータメンバとして持つ数値配列のアドレスを、
　この内部ポインタでポイントすると、ポインタ演算ができます。

・内部ポインタは、ポインタを機能拡張したものですから、
　ネイティブ側のデータ型には対応していますが、
　参照クラスには対応していません。

・また、内部ポインタは、スタック上に配置されるため、
　ローカル変数、引数、戻り値としては宣言できますが、
　データメンバとして宣言することはできません。

・テンプレート引数「**T**」に指定できるデータ型は、
　ネイティブ側のデータ型、または、
　基本データ型と互換性のあるプリミティブ型だけです。

　(一応、追跡ハンドルも指定はできますが、意味はありません。)

・いくつか試してみましょう。↓

```
int var1 = 140; // ネイティブのローカル変数を宣言する。
interior_ptr<int>  p1 = &var1; // そのｱﾄﾞﾚｽを、内部ﾎﾟｲﾝﾀに格納する。

int array1[] = {0,1,2}; // ネイティブ配列を宣言する。
interior_ptr<int>  p2 = array1; // そのｱﾄﾞﾚｽを、内部ﾎﾟｲﾝﾀに格納する。

int* p_array1 = new int[3]; // ネイティブ配列を動的に確保する。
```

```
interior_ptr<int>  p3 = p_array1; // そのｱﾄﾞﾚｽを、内部ﾎﾟｲﾝﾀに格納する。

Class1* p_obj1 = new Class1(); // ネイティブクラスをインスタンス化する。
interior_ptr<Class1> p4 = p_obj1; // そのｱﾄﾞﾚｽを、内部ﾎﾟｲﾝﾀに格納する。
Class1* p5 = p4; // ( ※ いったん、元のﾃﾞｰﾀ型のﾎﾟｲﾝﾀに戻さないとﾀﾞﾒ。)
p5->SetValue(120); // メソッドを呼び出せる。

// ネイティブクラスのインスタンス配列を、動的に確保する。
Class1* p_obj_array1 = new Class1[3];
interior_ptr<Class1> p6 = p_obj_array1; // そのｱﾄﾞﾚｽを、内部ﾎﾟｲﾝﾀに格納する。
(p6 + 1)->SetValue(120); // ﾎﾟｲﾝﾀ演算も可能。
```

・引数や戻り値にも使えます。↓

```
interior_ptr<int> Func1( interior_ptr<int> p1_ )
{
    return p1_;
}

void main()
{
    int v1 = 120; // ﾈｲﾃｨﾌﾞのﾛｰｶﾙ変数を宣言する。
    interior_ptr<int> p1 = &v1; // そのｱﾄﾞﾚｽを、内部ﾎﾟｲﾝﾀに格納する。
    interior_ptr<int> p2 = Func1( p1 ); // これもOK。
}
```

・データメンバを、内部ポインタで操作する場合は、次のように書きます。↓

```
// 参照構造体の宣言。
ref struct RefStruct1
{
    Int32 m1;
};

void main()
{
    // 参照構造体をｲﾝｽﾀﾝｽ化する。
   RefStruct1^ p_s1 = gcnew RefStruct1();

    p_s1->m1 = 100; // ﾃﾞｰﾀﾒﾝﾊﾞを、初期化しておく。

    // ﾃﾞｰﾀﾒﾝﾊﾞのｱﾄﾞﾚｽを、固定ﾎﾟｲﾝﾀに格納する。
    interior_ptr<Int32> p1 = &(p_s1->m1);

   *p1 = 200; // 固定ﾎﾟｲﾝﾀに、別の値を代入する。

   // ﾃﾞｰﾀﾒﾝﾊﾞの値も変更されている。
    Console::WriteLine( "現在値=" + p_s1->m1 );

}
```

固定ポインタ **pin_ptr<T>**

・通常、マネージヒープに置かれたインスタンスは、
ガベージコレクションによって移動させられる可能性があるため、
通常のポインタでは操作できないのですが、

今回紹介する「固定ポインタ」(ピンポインタ) を使うと、
ガベージコレクションに無視されるようになるため、
追跡ハンドルがポイントしているアドレスを、
ポインタに代入して、ポイントさせることができるようになります。

```
Int32^ p1 = gcnew Int32( 123 );
pin_ptr<int> p2 = &static_cast<int>( p1 );
int* p3 = p2;
int v1 = *p3;
```

・この固定ポインタは、ネイティブ側の関数やメソッドに、
マネージ系クラスのデータメンバへのポインタを渡すのに使います。

・固定ポインタがポイントしているインスタンスは、
依然としてマネージヒープ領域上にあるのですが、
ガベージコレクションから無視されているため、
通常のポインタと同じようにあつかうことができるのです。

・固定ポインタは、スタック領域上に置かれるローカル変数、
引数、戻り値には使えますが、ヒープ領域上に置かれるクラスの
データメンバとして持つことはできません。

・内部ポインタと同じで、ネイティブ側のデータ型にしか対応していません。↓

```
int var1 = 140; // ネイティブのローカル変数を宣言する。
pin_ptr<int>  p1 = &var1; // そのｱﾄﾞﾚｽを、固定ﾎﾟｲﾝﾀに格納する。
```

```
int array1[] = {0,1,2}; // ネイティブ配列を宣言する。
pin_ptr<int>  p2 = array1; // そのｱﾄﾞﾚｽを、固定ﾎﾟｲﾝﾀに格納する。

int* p_array1 = new int[3]; // ネイティブ配列を動的に確保する。
pin_ptr<int>  p3 = p_array1; // そのｱﾄﾞﾚｽを、固定ﾎﾟｲﾝﾀに格納する。

Class1* p_obj1 = new Class1(); // ネイティブクラスをインスタンス化する。
pin_ptr<Class1> p4 = p_obj1; // そのｱﾄﾞﾚｽを、固定ﾎﾟｲﾝﾀに格納する。
Class1* p5 = p4; // ( ※ いったん、元のﾃﾞｰﾀ型のﾎﾟｲﾝﾀに戻さないとﾀﾞﾒ。)
p5->SetValue(120); // メソッドを呼び出せる。

// ネイティブクラスのインスタンス配列を、動的に確保する。
Class1* p_obj_array1 = new Class1[3];
interior_ptr<Class1> p6 = p_obj_array1; // そのｱﾄﾞﾚｽを、固定ﾎﾟｲﾝﾀに格納する。
(p6 + 1)->SetValue(120); // ﾎﾟｲﾝﾀ演算も可能。
```

・データメンバを固定ポインタで操作する場合は、次のように書きます。↓

```
// 参照構造体の宣言。
ref struct RefStruct1
{
   Int32 m1;
};

void main()
{
    // 参照構造体をｲﾝｽﾀﾝｽ化する。
   RefStruct1^ p_s1 = gcnew RefStruct1();

    p_s1->m1 = 100; // ﾃﾞｰﾀﾒﾝﾊﾞを、初期化しておく。
```

```
    // ﾃﾞｰﾀﾒﾝﾊﾞのｱﾄﾞﾚｽを、固定ﾎﾟｲﾝﾀに格納する。
    pin_ptr<Int32> p1 = &(p_s1->m1);

   *p1 = 200; // 固定ﾎﾟｲﾝﾀに、別の値を代入する。

   // ﾃﾞｰﾀﾒﾝﾊﾞの値も変更されている。
    Console::WriteLine( "現在値=" + p_s1->m1 );

}
```

IntPtr構造体

・この構造体は、マネージヒープとネイティブヒープとで、
文字列や配列をやり取りするのに使います。

・「**Int**」というのは、「**Integer**」(整数)のことで、
要するにこの構造体は、標準整数の値を包んだものです。

・**Windows API** では、アドレスを、標準整数値にキャストして、
ウィンドゥの固有情報のところに持たせていましたが、
これと同じで、この構造体が持っている整数値もアドレスです。

・そのデータ型は、**32bit**システムでは int になり、
64bitシステムでは __int64 になります。

・**ToPointer**メソッドは、整数値を void* にキャストして返してくれます。

・その他のメンバについては、こちらをご覧ください。↓

→ IntPtr構造体 (msdn)

ネイティブ関数に値型のアドレスを渡す

```
Int32 v1 = 128; // 値型のﾛｰｶﾙ変数を宣言する。

pin_ptr<int> p_pin1 = &v1; // 変数のｱﾄﾞﾚｽを、固定ﾎﾟｲﾝﾀに代入する。

int* p1 = p_pin1; // 固定ﾎﾟｲﾝﾀを、ﾎﾟｲﾝﾀに代入する。
```

ネイティブ関数にマネージ文字列を渡す

```
using namespace System; // Console::WriteLine

#include <locale.h> // setlocale
#include <wchar.h> // wsprintf_s
#include <vcclr.h>   // PtrToStringChars

void main()
{
    // 「ﾛｹｰﾙ」には、日本語版Windowsの標準ｺｰﾄﾞｾｯﾄ「S-JIS ( cp932 )」を設定する。
    setlocale( LC_ALL , "Japanese_Japan.932" ); // ※ これがないと、全角文字が化ける。

    String^ p_text1 = gcnew String( L"あいうえお" ); // 文字列を作成する。

    // 文字列の追跡ハンドルを、内部ポインタに変換する。
    interior_ptr<const wchar_t> p_int1 = PtrToStringChars( p_text1 );
    Console::WriteLine( *( p_int1 + 1 ) ); // マネージメソッドに渡す。

    // 文字列の追跡ハンドルを、内部ポインタに変換する。
    pin_ptr<const wchar_t> p_pin1 = PtrToStringChars( p_text1 );
    ::wsprintf_s( L"%s¥n", p_pin1 ); // ネイティブ関数に渡す。
}
```

・Marshalクラスを使う場合は、次のように書きます。↓

```
using namespace System;
using namespace System::Runtime::InteropServices; // Marshal

#include <iostream>
using namespace std;
```

```
void main()
{
   std::wcout.imbue(  std::locale( "japanese" ) ); // これがないと化ける。

   String^ p_text1 = gcnew String( L"あいうえお" );

   IntPtr p_int1 = Marshal::StringToHGlobalUni( p_text1 );

   wchar_t* p_text2 = static_cast<wchar_t*>( p_int1.ToPointer() );

   std::wcout << p_text2 << std::endl;

   Marshal::FreeHGlobal( p_int1 ); // 必ず解放する。
}
```

・前回は、**C**言語の標準関数で出力しましたので、
今回は、**C++**の標準**IO** (STL) で出力してみました。

・「**StringToHGlobalUni**」メソッドは、**Unicode**用で、
その他にも、次のようなバリエーションがあります。↓

メソッドの名称	対応する文字列の種類
StringToHGlobalAnsi	char
StringToHGlobalUni	wchar_t
StringToHGlobalAuto	TCHAR
StringToHGlobalBstr	BSTR

・それと、**Marshal**クラスで**IntPtr**を変換する場合は、上の例のように、
ポインタを使わなくなった時点で、「**FreeHGlobal**」メソッドを呼んで、
ポインタを解放する必要があります。

・ただし、マネージヒープに確保した場合 (後述) は、ガベージコレクションが

勝手に解放してくれますので、不要です。

ネイティブ関数にマネージ配列を渡す

・ネイティブ配列とマネージ配列には互換性がないため、
当然ながら、追跡ハンドルを、強引にポインタにキャストしたり
するようなことはできません。

・しかし、単一値には互換性がありますから、
必要な領域をネイティブヒープ上に確保して、
要素値をコピーすることなら可能です。↓

```
using namespace System;
using namespace System::Runtime::InteropServices;

// ネイティブ関数の実装。( 内容はテキトー。)
void Func1( int* p_buf_, const size_t buf_length_ )
{
    for ( size_t i = 0; i < buf_length_; i++ )
    {
        Console::WriteLine( p_buf_[i] );
    }
}

void main()
{
    array<int>^ p_array1 = gcnew array<int>{ 0, 1, 2 };

    // sizeof は、size_t を返すが、AllocHGlobal ﾒｿｯﾄﾞの引数は、Int32 なので、
    // ｼｽﾃﾑ環境によっては、ｹﾀ落ちの警告が出てしまうことがある。
    #pragma warning( disable:4267 ) // この警告を無効にする。

    size_t buf_size = sizeof(int) * p_array1->Length; // ｻｲｽﾞを算出しておく。

    IntPtr p_buf1 = Marshal::AllocHGlobal( buf_size ); // ﾋｰﾌﾟ領域を確保する。

    // 配列をｺﾋﾟｰする。(※ このメソッドは、数値配列にのみ対応しています。)
```

```
    Marshal::Copy( p_array1, 0, p_buf1, p_array1->Length );

    int* p_buf2 = static_cast<int*>( p_buf1->ToPointer() ); // ﾎﾟｲﾝﾀに変換する。

    Func1( p_buf2, p_array1->Length ); // ﾈｲﾃｨﾌﾞ関数にﾎﾟｲﾝﾀを渡す。

    Marshal::FreeHGlobal( p_buf1 ); // 必ず解放する。

}
```

・このCopyメソッドでコピーできるのは、数値配列だけです。

・数値系の値クラスやマネージ配列は、内部形式が同じですから、
そのままコピーをすることができるのですが、
数値ではない Char、String、Boolean は、コピーできません。

マネージ関数に数値ポインタを渡す

```
// ﾏﾈｰｼﾞ側の関数
interior_ptr<int> Func1( interior_ptr<int> p1_, const size_t length_ )
{
   for ( size_t i = 0; i < length_; i++ )
   {
      // マネージ側のメソッドに渡しています。
      Console::WriteLine( "引数値=" + *p1_ );
      // ( 加算演算子によって、文字列が連結されると、
      //   Stringクラスのインスタンスが生成され、
      //    文字列がコピーされる。)
   }

   return p1_; // 戻り値として返すこともできます。
}

void main()
{
   int* p_array1 = new int[3]{0,1,2}; // ﾈｲﾃｨﾌﾞ配列を作成する。

   int* p1 = Func1( p_array1, 3 ); // ﾏﾈｰｼﾞ関数の呼び出し。
}
```

マネージ関数にネイティブ文字列を渡す

```
 wchar_t* p_text1 = L"あいうえお";
IntPtr p_int1( p_text1 );
String^ p_text2 = Marshal::PtrToStringUni( p_int1 );
Console::WriteLine( p_text2 );
```

もしくは、

```
 wchar_t* p_text1 = L"らりるれろ";
String^ p_text2 = gcnew String( p_text1 );
Console::WriteLine( p_text2 );
```

マネージ関数に配列を渡す

・ネイティブ配列(へのポインタ)を、マネージの関数に渡す場合は、
　マネージ配列のインスタンス化と、要素値のコピーを行います。 ↓

```
int array1[] = {0,1,2}; // ﾈｲﾃｨﾌﾞ配列を宣言する。 (ﾛｰｶﾙ変数)

IntPtr p_ptr1( array1 );

array<int>^ p_array2 = gcnew array<int>( sizeof( array1 ) / sizeof( int ) );

Marshal::Copy( p_ptr1, p_array2, 0, p_array2->Length );

for each ( int item in p_array2 )
{
	Console::WriteLine( item );
}
```

ネイティブ連携のまとめ (早見表)

マネージからネイティブへ　(CLI → C++)

	元のﾃﾞｰﾀ型	仲介する特殊ﾎﾟｲﾝﾀ	領域の再確保とｺﾋﾟｰ	ｷｬｽﾄ後のﾃﾞｰﾀ型	解放
数値	Int32	pin_ptr<int>	不要	int	不要
文字列	String^	IntPtr	必要	wchar_t*	必要
配列	array<Int32>^	IntPtr	必要	int [ ]	必要

ネイティブからマネージへ　(C++ → CLI)

	元のﾃﾞｰﾀ型	仲介する特殊ﾎﾟｲﾝﾀ	領域の再確保とｺﾋﾟｰ	ｷｬｽﾄ後のﾃﾞｰﾀ型	解放
数値	int	interior_ptr<int>	不要	Int32	不要
文字列	wchar_t*	IntPtr	必要	String^	不要
配列	int [ ]	IntPtr	必要	array<Int32>^	不要

・単一値については、内部ポインタと固定ポインタを利用します。
・文字列と配列については、相手側でヒープ領域を確保して、コピーを行う必要があります。
・そして、ネイティブヒープに領域を確保した場合は、解放をする必要があります。

基本データ型

C++	CX	意味
bool	bool	論理値。(8 bit 整数)
__wchar_t	char16	Unicode 文字。 (UTF-16 文字コードを表す 16 bit 整数。)
short	int16	16 bit 符号付き整数。
unsigned short	uint16	16 bit 符号なし整数。
int	int	32 bit 符号付き整数。
unsigned int	uint32	32 bit 符号なし整数
long long __int64	int64	64 bit 符号付き整数。
unsigned long long	uint64	64 bit 符号なし整数。
float	float32	32 bit 浮動小数点数。(IEEE 754 準拠)
double	float64	64 bit 浮動小数点数。(IEEE 754 準拠)

→ 詳細はこちら (msdn)

CLIからの変更点 (早見表)

名称	C++	CLI	CX
共有ポインタ	std::shared_ptr<T>	T^	CLIと同じ
参照	T&	T%	CLIと同じ
文字列	std::wstring	String^	CLIと同じ
配列	T 変数名[要素数] std::array<T>	array<T>	CLIと同じ
NULL許容型	boost::optional<T>	System::Nullable<T>	platform::IBox<T>
new 演算子	new	gcnew	ref new
delete 演算子	delete	解放はｶﾞﾍﾞｺﾚ任せ。 deleteも使える。	ﾃﾞｽﾄﾗｸﾀを呼び出す (参照ｶｳﾝﾄが減る。)
構造体	struct s1 {};	value struct s1 {}; value class s1 {};	CLIと同じ
クラス	class c1 {};	ref class c1 {}; ref struct s1 {};	CLIと同じ
列挙体	enum e1{};	enum class e1 {}; enum struct e1 {};	CLIと同じ
インターフェイス	__interface i1 {};	interface class i1 {}; interface struct i1 {};	CLIと同じ
デリゲート	std::function	delegate T f1( );	CLIと同じ
イベント	なし	event D e1;	CLIと同じ
プロパティー	なし	property p1{};	property T p1; property T p1[n]; property T default[n];
テンプレート	template<typename T>	generic<typename T>	CLIと同じ