C++/CLI Tutorial

Author: Adam Sawicki, adam__DELETE__@asawicki.info, www.asawicki.info
Version 1.0, December 2011

Table of Contents

Introduction

Welcome to my C++/CLI tutorial. I'm not sure whether it can be called a proper tutorial, but anyway my intention was to write an introductory article from which you can learn basics of this language. I've been using C++/CLI during past 2 years in my previous work. Not that it was my decision to use this technology, but I now think my boss was right in choosing C++/CLI for the kinds of projects we did in the company. C++/CLI is a nice language and has some unique features, so I think it's worth knowing. This article is far from being comprehensive or systematic. It's more practice oriented, based on my experiences in developing quite complex software in C++/CLI, so I described here only these language features that I know well and I've found useful in my code, not all what is available in the language syntax.

If you ask about a difficulty level of this text, I'd answer that it's intermediate. It's not advanced as it only introduces basics of a programming language, but on the other hand I believe that to understand C++/CLI you must already know both native programming in C++ (including subjects such as headers, pointers, classes) as well as .NET (including knowledge about garbage collector, Windows Forms, .NET standard library).

What is C++/CLI?

C++/CLI is a separate programming language which can be viewed as an extension to C++ syntax, but at the same time it's one of the programming languages of .NET platform, just like C# or Visual Basic .NET. It's designed and implemented by Microsoft, so it's not portable to Linux or other systems. It requires Microsoft Visual C++ IDE to compile and debug code, so we have no alternative compilers like gcc or icc available for this language. Good news is that the language is supported in Visual Studio 2005, 2008 and 2010, including corresponding versions of the free Visual C++ Express Edition.

To run programs written in C++/CLI, just like for other .NET applications, user must have appropriate version of the free Microsoft .NET Framework installed in his Windows. I didn't succeed in running a C++/CLI program in Linux, neither using Wine nor Mono.

Why Use C++/CLI?

Why learn and use such weird hybrid language if we have C# with nice syntax, designed specifically for .NET platform? C++/CLI has an unique feature among other .NET languages: You can freely mix managed (.NET) and native (C++) code in a single project, single source file, even single function code. It makes the language hard to replace in some applications, like:

- When you write software that needs to use some native as well as managed libraries.
- When you write a library that links native with managed code, for example exposes an interface of some native library to .NET code.
- When you write a program that needs both efficient processing of some binary data (which is best done in native C++ code using pointers) and has complex graphical interface (which is best done using Windows Forms) – just like I did in my job.

What C++/CLI is Not?

You may think that C++/CLI is some ugly, proprietary hack to C++ introduced by bad "M$" to confuse programmers. That's not exactly true. Microsoft put effort to make this language really good. Some famous computer scientists and

C++ gurus were involved in its development, like Stanley B. Lippman and Herb Sutter. The language itself is standardized as ECMA-372.

I must also clarify that C++/CLI is not the same as the old language called Microsoft Extensions for C++. That strange language which extended C++ syntax with ugly keywords like `__gc` or `__value` was predecessor of C++/CLI and is now deprecated. The syntax of C++/CLI is totally different and much more pleasant, as you will see in the next chapters.

Hello World Example

It's time for the first example code. You can find it in attached archive as `Hello_world` subdirectory. I coded all examples for this tutorial in Visual C++ 2010 Express. Traditionally first example will be a simple application that prints "Hello World" text on its output. To make it I've started Visual C++ Express and created new project of type `CLR Console Application`. So it is a project in C++/CLI that compiles to an EXE file. It's .NET application – it requires .NET Framework installed. It shows system console and no windows. Project contains some automatically generated files, including `AssemblyInfo.cpp` where you can fill in some metadata about your program like product name and version. But the main source file is `Hello_world.cpp` with following contents:

```
#include <iostream>

int main(array<System::String ^> ^args)
{
  System::Console::WriteLine(L"Managed Hello World");
  std::cout << "Native Hello World" << std::endl;
  return 0;
}
```

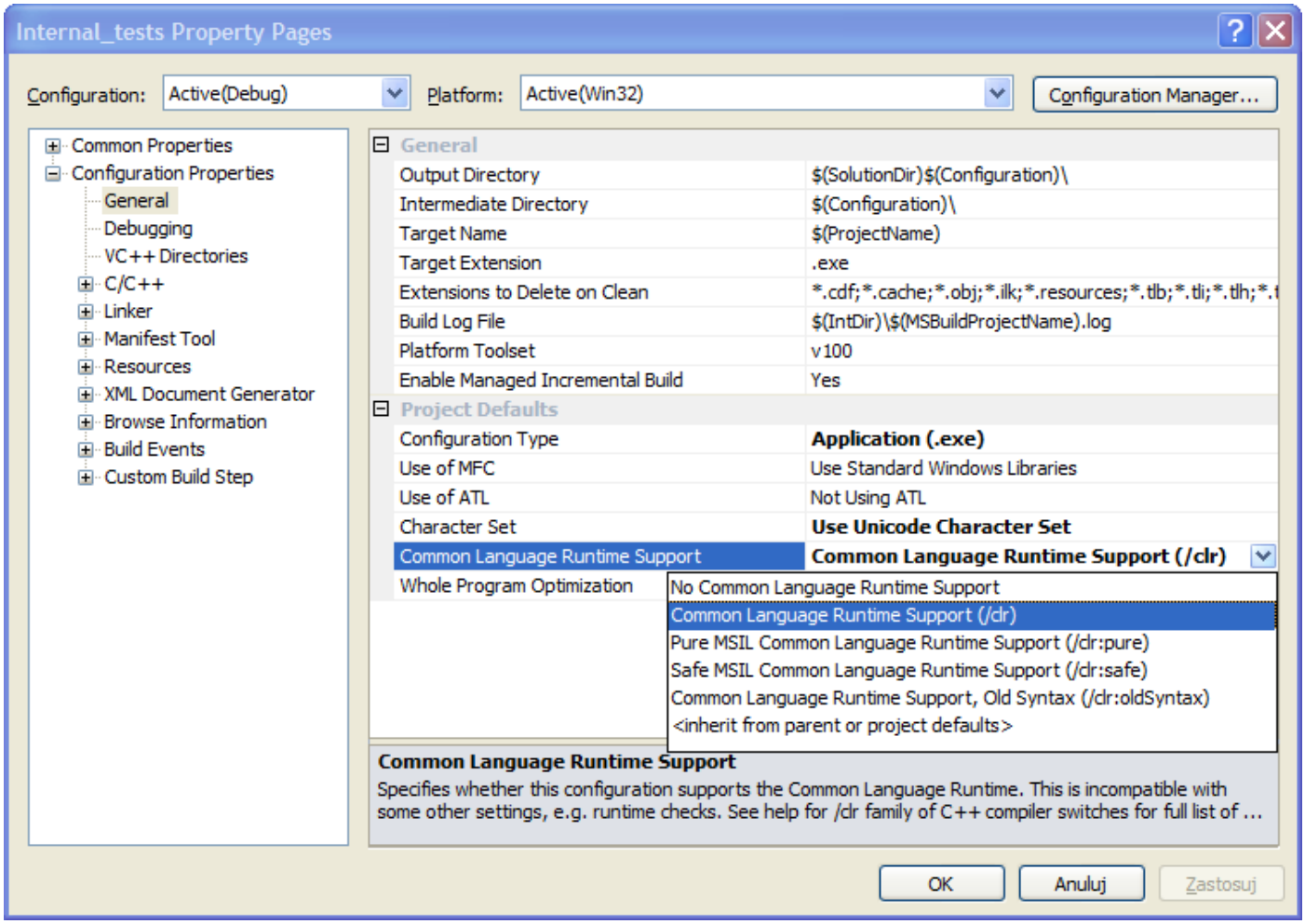
This example shows two messages. First one is done using traditional .NET way – `WriteLine` static method of `System.Console` class. Second is done using method recommended for C++ - `std::cout` stream from standard C++ library, which required `#include <iostream>`. As you can see, managed and native code are freely mixed here in a single function code. That's the biggest power of C++/CLI! Of course there are more strange things here that can seem confusing to you, but don't worry – I'll explain them later. Right now here is the output of this program:

```
Managed Hello World
Native Hello World
```

Project Properties

It's time to go deeper into some details of the language syntax. If you already know C++ and/or some .NET language and Visual Studio IDE, I hope you know how to get to the Project Properties window. Project properties look totally different in native C++ and in .NET. The window you see when coding in C++/CLI is like in native C++ project.

There is one thing about it I must emphasize at the beginning: an option in `Configuration Properties / General` branch called `Common Language Runtime Support`. This option decides whether the compiler will treat and compile your code as it is in native C++ or C++/CLI, but there are several options for C++/CLI, all starting from `/clr`. The one you should choose is `/clr` and NOT the `/clr:pure`. I'm not sure what they do :) but switching this option as described helped me many times fixing some strange compiler/linker errors, like the hated "Unresolved external symbol…".



Namespaces

Namespaces work similar way in native and managed code. In C++/CLI they have a syntax like in C++ and can be freely mixed. There is no distinguish between native and managed namespaces, like you will see for classes. Here is a small example of defining a namespace (`MyNamespace`), qualifying identifier with a namespace (`MyNamespace::DoEverything`) and importing a namespace with `using namespace` directive:

```
#include <iostream>

using namespace System;
using namespace std;

namespace MyNamespace
{
  void DoEverything()
  {
    Console::WriteLine(L"Managed Hello World"); // System::Console
    cout << "Native Hello World" << endl; // std::cout, std::endl
  }
}

int main(array<System::String ^> ^args)
{
  MyNamespace::DoEverything();
  return 0;
}
```

Classes and Pointers

Here comes the biggest piece of knowledge you have to learn to use C++/CLI – the one about classes, objects, pointers, references, destructors, garbage collector etc. Don't worry – I will explain everything step by step, show some simple examples and summarize everything at the end.

First thing you need to know is that there is strict distinction in C++/CLI between native and managed classes and that some rules apply to some specific types. There are also separate pointer/reference operators for these two types. Pointers to native objects work like in native C++. You have to free them manually – they are not garbage collected. Objects of managed classes, on the other hand, behave like in C# and the rest of .NET platform – you have to allocate them dynamically on the heap and they are automatically freed by the garbage collector. There is a completely new syntax in C++/CLI for these managed objects that you will see soon.

Native Classes

Let's start with something you already know – native C++ classes. You can define them and use them just like there was no managed part, only the old good native C++. You can, for example, define a class like this:

```
class NativeMonster
{
public:
  NativeMonster(int HP) : m_HP(HP) {
    cout << "NativeMonster Constructor" << endl;
  }
  ~NativeMonster() {
    cout << "NativeMonster Destructor" << endl;
  }
  void TellHitPoints() {
    cout << "NativeMonster has " << m_HP << " HP" << endl;
  }
private:
  int m_HP;
};
```

`NativeMonster` is a native class. It has constructor, destructor, a private field `m_HP` and a public method `TellHitPoints`. You can use it in some code by creating a local object of this type by value, on the stack, like this:

```
int main(array<System::String ^> ^args) {
  NativeMonster stack_monster(100);
  stack_monster.TellHitPoints();
}
```

This program prints following output:

```
NativeMonster Constructor
NativeMonster has 100 HP
NativeMonster Destructor
```

You can also allocate native objects on the heap, using standard C++ operator `new`. Always remember to free them with operator `delete` when no longer needed because native objects are not garbage collected – forgetting to free them causes memory leaks! Following program prints same output, but here the object of type `NativeMonster` is allocated dynamically on the heap and used by pointer, not by value.

```
int main(array<System::String ^> ^args) {
  NativeMonster *monster_pointer = new NativeMonster(100);
  monster_pointer->TellHitPoints();
  delete monster_pointer;
}
```

As you can see, native classes can be coded in C++/CLI same way as in native C++. You use class keyword, implement constructor, destructor, fields and methods (or not, depending on whether you need them) and then you can create objects of this class in the stack or on the heap, allocate them with `new`, free them with `delete` and reference them with pointers. You use `*` operator for declaring pointers and pointer dereference, `&` operator for declaring references and getting object address (not shown in the example) and `->` operator to access object members from a pointer – all exactly like in C++.

Native structures look similarly so I won't show them here. Just like in C++ you can define structures using `struct` keyword and then use them by value or by native `*` pointer.

Managed Classes

Now it's time to see how a managed class looks like in C++/CLI. Here is an example:

```
ref class ManagedMonster
{
public:
  ManagedMonster(int HP) : m_HP(HP) {
    cout << "ManagedMonster Constructor" << endl;
  }
  void TellHitPoints() {
    cout << "ManagedMonster has " << m_HP << " HP" << endl;
  }
private:
  int m_HP;
};
```

First thing that strikes the eye here is that a special new keyword `ref class` is used. It tells the compiler that we define a managed class. Rest is the same – you can write constructor and use C++ initialization list inside it, you can define fields and methods, you can use `private`, `protected` and `public` keywords. You can also call native functions (like using `std::cout` shown here) or managed function (like printing to console using `System::Console::WriteLine`) no matter if you are inside a body of a native class or managed class method.

Objects of managed classes are never created on the stack but only on the heap. A new type of "pointer" – a reference to a managed object, uses `^` character in C++/CLI. It is a managed heap this time so different rules apply to allocation and deallocation of such objects. We allocate them using `gcnew` keyword. There is no `delete` operator because managed objects are automatically freed by the garbage collector –in some unspecified future, possibly on another background thread, but always after such objects is no longer referenced by any pointer. For example:

```
ManagedMonster ^monster_ref = gcnew ManagedMonster(120);
monster_ref->TellHitPoints();
```

This example prints following output:

```
ManagedMonster Constructor
ManagedMonster has 120 HP
```

If you want to clear the pointer, just assign null to it. But that's not a normal "null", you cannot use `NULL` macro or `0` here. You have to do it with special `nullptr` keyword, which is an empty value of type compatible with managed pointers.

```
monster_ref = nullptr;
```

`nullptr` was introduced as keyword in C++/CLI. You can also use it as null value of native pointers. when coding in C++/CLI. But same keyword is also being introduced to the new native C++11 standard (formerly known as C++0x) as a replacement of `0` or `NULL` for pointers, so you can also use it in native C++.

Managed Structures

Difference between classes and structures in native C++ is very small – structures have public members and public inheritance by default unless you explicitly state otherwise, while classes default to private. In C#, on the other hand, structures are very different – they have limited functionality and they are passed by value, not by reference. C++/CLI supports the former with `class` and `struct` keywords, as well as the latter by introducing the `ref` and `value` keywords. So to define a managed structure in C++/CLI that will be stored by value like the `System::DateTime` or `System::Drawing::Color` type, use `value class` or `value struct`. For example:

```
value class Position {
public: float x, y, z;
};

int main(array<System::String ^> ^args) {
  Position pos;
  pos.x = 0.0f;
  pos.y = 1.0f;
  pos.z = 7.0f;
  Console::WriteLine(L"The position is ({0},{1},{2})", pos.x, pos.y, pos.z);
}
```

So to summarize the available types of classes and structures in C++/CLI, let's see a table:

Keyword	Domain	Default Access	Equivalent	Used by
`class`	Native	private	class in C++	Value, native pointer or reference
`struct`	Native	public	struct in C++	Value, native pointer or reference
`ref class`	Managed	private	class in C#	Managed reference
`ref struct`	Managed	public	class in C#	Managed reference
`value class`	Managed	private	struct in C#	Value
`value struct`	Managed	public	struct in C#	Value

Managed Class Destructors

Despite managed objects being automatically freed by garbage collector, they can have destructors. What is more, there are two kinds of such special methods in C++/CLI – a destructor and a finalizer. They have to be used whenever:

- Your class keeps some resources that will not be freed by garbage collector but have to be freed manually, like a native pointer or an object that needs to have some `Close()` method called before destruction.
- Your class keeps some resources that should not stay acquired for as long as garbage collector wants but should be freed right after they are no longer needed because they occupy a lot of memory or keep lock of some system resources, like a bitmap, opened file, network socket or database connection.

Finalizer – declared as `!ClassName();` – is equivalent to overriding `Object::Finalize` method and to the class destructor in C#. It is called by garbage collector right before an object is freed from memory. Such call can be made in some unspecified future and on a separate background thread.

Destructor on the other hand – declared like C++ destructor as `~ClassName();` – is equivalent to implementing `IDisposable` interface and its `Dispose` method, as well as calling `GC::SuppressFinalize` at the end. It all happens automatically so you don't have to explicitly inherit from `IDisposable` or anything. All you need to do is to define a destructor.

Remember that this managed destructor – just like `Dispose` method you probably know from C# – is not guaranteed to be called. It is available for the user of your class if he decides to explicitly order your object to free its resources in

some specific moment, but if he does not, then you have to do all the necessary cleanup in the finalizer. Here is an example:

```
ref class ManagedMonster {
public:
  ManagedMonster(int HP);
  ~ManagedMonster();
  !ManagedMonster();
  void TellHitPoints();
private:
  int *m_DynamicHP;
};

ManagedMonster::ManagedMonster(int HP) : m_DynamicHP(new int(HP)) {
  cout << "ManagedMonster Constructor" << endl;
}

ManagedMonster::~ManagedMonster() {
  cout << "ManagedMonster Destructor" << endl;
  this->!ManagedMonster();
}

ManagedMonster::!ManagedMonster() {
  cout << "ManagedMonster Finalizer" << endl;
  delete m_DynamicHP;
}

void ManagedMonster::TellHitPoints() {
  cout << "ManagedMonster has " << *m_DynamicHP << " HP" << endl;
}
```

The class in this example has constructor, destructor and finalizer. There are multiple ways of using it. First way – the one you have already seen goes like this:

```
int main(array<System::String ^> ^args) {
  ManagedMonster ^monster_ref = gcnew ManagedMonster(120);
  monster_ref->TellHitPoints();
}
// Garbage Collector frees allocated object at the end of the program.
```

Output of this program will be:

```
ManagedMonster Constructor
ManagedMonster has 120 HP
ManagedMonster Finalizer
```

As you can see, the destructor is not called, only the finalizer. That's because we don't make any use of `IDisposable` implementation here. We dynamically allocate the managed object as usual, use it and let the garbage collector to call its finalizer and free it at the end.

Next you will see two things that look very strange, so please pay attention to remember this syntax. In C# you order an object of a class that implements `IDisposable` interface to free its resources by calling `Dispose` method. C++/CLI equivalent is to use… `delete` operator on a managed object, like this:

```
int main(array<System::String ^> ^args) {
  ManagedMonster ^monster_ref = gcnew ManagedMonster(120);
  monster_ref->TellHitPoints();
  delete monster_ref; // Call Dispose.
}
```

The output of this program is:

```
ManagedMonster Constructor
ManagedMonster has 120 HP
ManagedMonster Destructor
ManagedMonster Finalizer
```

Using delete operator calls `IDisposable.Dispose` method, implemented in our code as class destructor. Calling this destructor automatically suppresses finalizer so the finalizer is not called before garbage collector frees the object – we have to call it manually from destructor, following the practice recommended by Microsoft. Main difference between this example and the previous example is that here we explicitly specify the moment where we want the finalizer to be called, while in the previous example garbage collector is free to call finalizer in some distant future, leaving some possibly heavy resources (like opened file, database connection or… a dynamically allocated native `int` in our case) locked and loaded for a longer time.

Another, more convenient way of calling `Dispose` in C# is the `using` keyword. We can enclose construction of an `IDisposable` object in the `using(){}` section and the `Dispose` method will be automatically called at the end of its scope. C++/CLI has equivalent mechanics but it's syntax is also weird. To do this you need to define an object of our managed class by value, not by managed pointer `^` - like it was constructed on the stack, although managed objects cannot really be on the stack, they are always dynamically allocated from the managed heap. When an object is defined like this, its destructor is automatically called when we go out of the current scope.

```
int main(array<System::String ^> ^args) {
  ManagedMonster monster(120);
  monster.TellHitPoints();
} // End of scope calls Dispose.
```

Output of this program is same as the previous one.

Pointers and References

As you already seen, we have two types of pointers available in C++/CLI. First are native pointers. They look and behave exactly like in native C++, so you allocate objects with `new` operator, you must manually free them with `delete` operator and you use operators `*` and `&`. Second type of pointers are references to managed objects. We allocate them with `gcnew` operator, don't have to free them as they are tracked by the garbage collector and the operator for them is `^`. The only thing that is missing there is the opposing operator for managed pointers, like `&` for native pointers that gets object address or defines a reference. There actually is such operator and it looks like this: `%`. Example:

```
void AskMonster(ManagedMonster ^monster) {
  monster->TellHitPoints();
}

int main(array<System::String ^> ^args) {
  ManagedMonster ^monster_1 = gcnew ManagedMonster(100);
  AskMonster(monster_1);
  ManagedMonster monster_2(200);
  AskMonster(%monster_2);
}
```

This program prints following output:

```
ManagedMonster Constructor
ManagedMonster has 100 HP
ManagedMonster Constructor
ManagedMonster has 200 HP
```

Another application of the % operator is to pass a parameter to a function by reference, to be able to return a new value. It is so called output parameter. For example:

```
void Add(int %result, int a, int b) {
  result = a + b;
}

int main(array<System::String ^> ^args) {
  int result, a = 2, b = 3;
  Add(result, a, b);
  Console::WriteLine(L"{0} + {1} = {2}", a, b, result);
}
```

This program prints following output:

```
2 + 3 = 5
```

To summarize this topic, let's see a table with all operators for pointers and references in C++/CLI:

Operation	Native Code	Managed Code
Pointer definition Pointer dereference	*	^
Reference definition Address-of	&	%
Member access	->	->
Allocation	new	gcnew
Deallocation	delete	delete (calls Dispose)

Rules of Containment

I've already shown that native and managed code can be freely mixed in a body of a single function. Unfortunately there is no such freedom when it comes to defining variables/fields of different types. Some restrictions apply here.

First, we cannot define global variables of managed types. That's probably because in .NET everything must lie inside classes.

```
NativeMonster g_GlobalNativeMonster(100); // OK
ManagedMonster ^g_GlobalManagedMonster = gcnew ManagedMonster(200); // ERROR
```

The compilation error we get here is:

```
error C3145: 'g_GlobalManagedMonster' : global or static variable may not have managed type
'ManagedMonster ^' may not declare a global or static variable, or a member of a native type that refers
to objects in the gc heap
```

Workaround for this limitation is to define a managed class with public, static member of the managed type we need as a global variable – like this:

```
ref class Globals {
public:
  static ManagedMonster ^GlobalManagedMonster = gcnew ManagedMonster(200);
};
```

Second, managed class can have members of native pointer types, but native classes cannot contain managed objects. That's probably because it would be hard for garbage collector to track the location and lifetime of a managed pointer if it would be contained in a native data structure.

```
ref class ManagedDungeonRoom {
  NativeMonster *m_Goblin; // OK
```

```
  ManagedMonster ^m_Boss;  // OK
};

class NativeDungeonRoom {
  NativeMonster *m_Goblin; // OK
  ManagedMonster ^m_Boss;  // ERROR
};
```

The error we get here is:

```
error C3265: cannot declare a managed 'm_Boss' in an unmanaged 'NativeDungeonRoom' may not declare a
global or static variable, or a member of a native type that refers to objects in the gc heap
```

Fortunately there is a solution to this. You need to use a special smart pointer class: `msclr::gcroot` from `<msclr\gcroot.h>` header. Here is an example:

```
#include <msclr\gcroot.h>

class NativeDungeonRoom {
private:
  NativeMonster *m_Goblin; // OK
  msclr::gcroot<ManagedMonster^> m_Boss; // Also OK
public:
  NativeDungeonRoom()
    : m_Goblin(new NativeMonster(10))
    , m_Boss(gcnew ManagedMonster(1000)) {
  }
  ~NativeDungeonRoom() {
    delete m_Goblin;
  }
  void AskMonsters() {
    m_Goblin->TellHitPoints();
    m_Boss->TellHitPoints();
  }
};

int main(array<System::String ^> ^args) {
  NativeDungeonRoom dungeon_room;
  dungeon_room.AskMonsters();
}
```

This program prints following output:

```
NativeMonster Constructor
ManagedMonster Constructor
NativeMonster has 10 HP
ManagedMonster has 1000 HP
NativeMonster Destructor
```

If you try to define member variables of different types by value, not by pointer, the result will be like this:

```
ref class ManagedDungeonRoom {
  NativeMonster m_Goblin; // ERROR
  ManagedMonster m_Boss;  // OK.
  Position m_Position;    // OK.
};

class NativeDungeonRoom {
  NativeMonster m_Goblin; // OK.
  ManagedMonster m_Boss;  // ERROR
```

```
  Position m_Position;    // OK.
};
```

`NativeMonster` cannot be contained inside `ManagedDungeonRoom` because of error:

```
error C4368: cannot define 'm_Goblin' as a member of managed 'ManagedDungeonRoom': mixed types are not
supported
```

While `ManagedMonster` cannot be contained inside `NativeDungeonRoom` because of these errors:

```
error C3265: cannot declare a managed 'm_Boss' in an unmanaged 'NativeDungeonRoom' may not declare a
global or static variable, or a member of a native type that refers to objects in the gc heap
error C3076: 'NativeDungeonRoom::m_Boss' : you cannot embed an instance of a reference type,
'ManagedMonster', in a native type
```

Additional Class Topics

You may be accustomed to this, but in fact C++ has quite weird syntax for object-oriented concepts. Method is abstract if defined as pure virtual `=0` while it still can but don't have to have a body, class is abstract automatically if it has some pure virtual methods, methods overridden in derived class can repeat `virtual` keyword but don't have to, there is no distinction for classes and interfaces… C#, on the other hand, has different syntax, with lots of keywords for such OOP concepts, like `abstract`, `override`, `new` etc. To merge these two worlds, designers of C++/CLI provided following syntax:

When a reference type inherits from another reference type, virtual functions in the base class must explicitly be overridden (with `override`) or hidden (with `new` – new slot in vtable). The derived class functions must also be explicitly marked as `virtual`.

```
ref class ManagedMonster {
  public: virtual int GetMaxHP() { return 0; }
};

ref class Boss : public ManagedMonster {
  public: virtual int GetMaxHP() override { return 1000; }
};
```

Interfaces can be defined using `interface` keywords, by typing `interface class` or `interface struct` (which mean the same). Interfaces are not full classes – they can contain only declarations of public functions, events and properties. They can also have static members.

```
interface class IMonster {
  public: int GetMaxHP();
};

ref class Boss : public IMonster {
  public: virtual int GetMaxHP() { return 1000; }
};
```

A class that does not implement all abstract methods declared in inherited classes and interfaces is automatically abstract. A method or whole class can also be forced to be considered abstract by using `abstract` keyword. Abstract class cannot be instantiated. Abstract method declared using `abstract` keyword is similar to a pure virtual method declared using `=0` syntax known from native C++.

```
ref class BaseMonster abstract {
  public: virtual void MakeNoise() abstract;
};
```

`sealed` keyword can be used for classes and methods. For classes it means that no other classes can inherit from this one.

```
ref class BossOfLevel1 sealed : public ManagedMonster {
  // ...
};
```

For methods the `sealed` keyword means that the virtual method cannot be further overridden in derived classes.

```
ref class Boss : public ManagedMonster {
  public: virtual int GetMaxHP() sealed { return 1000; }
};
```

Static class – a class that can contain only static members and cannot be instantiated – is defined in C++/CLI using `abstract sealed` keywords:

```
ref class Globals abstract sealed {
  public: static ManagedMonster ^g_GlobalMonster;
};
```

Last but not least, managed classes require declaration prior to usage and can have forward declarations, just like native C++ classes. So for example if you define a member variable of some managed class in another class in your code:

```
ref class ManagedDungeonRoom {
  NativeMonster *m_Goblin;
  ManagedMonster ^m_Boss;
};
```

Then you need to either `#include` header with definition of this class:

```
#include "ManagedMonster.h"
```

or forward declare this class, which is a good idea to do wherever possible because it reduces dependencies between headers and reduces compilation time:

```
ref class ManagedMonster;
```

Windows Example

I think now it's time for another, bigger example. It will be GUI application this time, not console one. You can find the source code and compiled EXE in the `Form_app` subdirectory of attached archive. To create this project I selected `New Project` command in Visual C++ 2010 and selected `CLR / Windows Forms Application`.

This created a project with some files already filled at start: resources containing default icon, `AssemblyInfo.cpp` file, precompiled header `stdafx.h`, `Form_app.cpp` source file with `main()` function and, what is most important here, an initial Windows Forms window split into two files: `Form1.h` (intended to be modified in text editor) and `Form1.resX` (manged by visual window designer). By right-clicking on the `Form1.h` node in `Solution Explorer` and selecting `View Designer`, you open visual form designer where you can design your window, put controls onto it using `Toolbox` panel and change their properties using `Properties` panel. Everything exactly like in C# and other .NET languages. By right-clicking on the `Form1.h` and selecting `View Code`, you open the `Form1.h` file in text editor. There you can see definition of `Form1` class with `InitializeComponent` method enclosed in `#pragma region`, filled with code that creates and initializes form controls you have designed.

Visual C++ creates only header file for the form. You could theoretically write all your code there, but what I like to do is creating corresponding `cpp` file and writing definitions of the class methods there, like we usually do in native

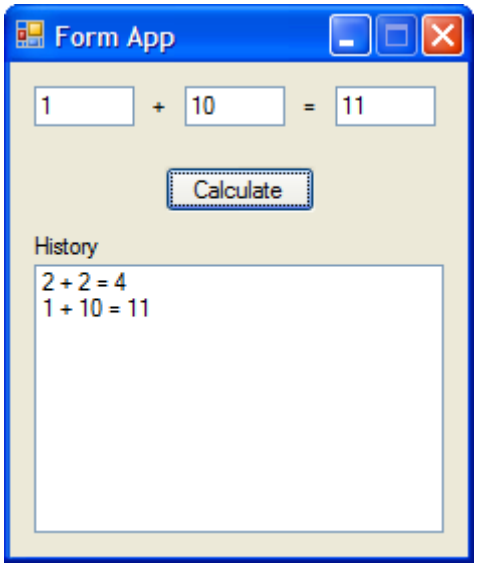
C++. To do it in this project I added `New Item` of type `C++ File (.cpp)` and named it `Form1.cpp`. I then moved definition of class constructor and destructor there, leaving only declarations in header file. So at this stage the content of the `Form1.cpp` file is:

```
#include "stdafx.h"
#include "Form1.h"

namespace Form_app {

Form1::Form1()
{
  InitializeComponent();
}

Form1::~Form1()
{
  if (components)
    delete components;
}

} // namespace Form_app
```

The logic I've put into this application is a simple calculator that performs addition of two numbers. To achieve this I used three TextBox controls (two for arguments, third for result), a button and some labels. Additionally I've added a "History" ListBox to add information about each operation performed to the list. The method that handles button click and performs all these operations contains following code:

```
System::Void Form1::button1_Click(System::Object^  sender, System::EventArgs^  e)
{
  try {
    double a = double::Parse(textBox1->Text);
    double b = double::Parse(textBox2->Text);
    double c = a + b;
    textBox3->Text = c.ToString();

    System::Text::StringBuilder sb;
    sb.AppendFormat(L"{0} + {1} = {2}", a, b, c);
    listBox1->Items->Add(sb.ToString());
  }
  catch (System::Exception ^ex) {
    MessageBox::Show(this, ex->Message, L"Error", MessageBoxButtons::OK, MessageBoxIcon::Error);
  }
}
```

Despite its simplicity a lot of things happen here. Many of them can be new to you. But don't worry – I'll explain all of them in following chapters of this tutorial.



Delegates and Events

There is one feature of a programming language that is especially useful when working with GUI. It's called delegates, events, sometimes signals, slots, callbacks or pointers to members. It is one of these things that modern, high level programming languages have, while native C++ does not. Delphi has it, C# and other languages of .NET platform also have it. You may say that C++ also has pointers to class methods in its syntax, but I mean something completely different. Pointers to members in C++ remember just an offset – point to some method of matching declaration but only inside given particular class and do not remember a particular object of this class. This make them not very useful. The pointers to members I mean here should be able to point to a PARTICULAR method with matching declaration but in a PARTICULAR object of ANY class. That's what is needed when we code a class representing a window and we want to implement some methods that will be called when a button on this form is pressed, a textbox is being changed etc.

To compensate for the lack of this feature, all GUI libraries for native C++ (like wxWidgets, Qt or MFC) provide their own solution for such pointers. There are also some independent libraries for this, like FastDelegate by Don Clugston or The Impossibly Fast C++ Delegates by Sergey Ryazanov. They work this way: each pointer to member holds two fields: a pointer to some object and an offset to method in its class.

Fortunately C++/CLI provides such mechanism just like whole .NET does in form of delegates and events. You can see the example usage in the code we talk about here, as method `Form1::button1_Click` reacting on clicking `button1`. Documentation says that `Button` class has `Click` event of type `event EventHandler^`, where `EventHandler` is defined as following delegate (using C++/CLI syntax):

```
delegate void EventHandler(Object^ sender, EventArgs^ e)
```

After double-clicking on the `button1` in form designer, Visual C++ creates the `button1_Click` method with signature compatible with this delegate and binds it to the `button1.Click` event. The syntax for registering method to react on some event can be found in automatically-generated code inside `InitializeComponent` method and looks like this:

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

As you can see, using events in C++/CLI requires:

1. Creating a delegate with `gcnew`. It takes two arguments – pointer to some object (`this`) and pointer to some method of its class that has signature compatible with the event (`Form1::button1_Click`).
2. Adding delegate to the event with `+=` operator.

Strings

There are different ways of handling character strings in C and C++. Programs written in C use `char*` type. Standard library of C++ defines `std::string` class. .NET library has its own built-in type for that – `System::String` class aliased in C# as just `string` keyword. We have all of this available in C++/CLI so we have to learn how to deal with such different kinds of strings and how to convert between them.

Native Strings

First, we can use old good null-terminated C strings of type `char*`. We can also `#include <string>` and use `std::string` class from C++ standard library (STL). Unicode versions (`wchar_t*` and `std::wstring` types) are also available to use. Here are example variable declarations and initialization:

```
const char    *native_sz   =  "Native ANSI C string";
const wchar_t *native_wsz  = L"Native Unicode C string";
std::string    native_str  =  "Native ANSI C++ string";
std::wstring   native_wstr = L"Native Unicode C++ string";
Console::WriteLine(L"The length of native_sz is {0}", strlen(native_sz));
Console::WriteLine(L"Comparing two strings returned {0}", wcscmp(native_wsz, native_wstr.c_str()));
```

This program produces following output:

```
The length of native_sz is 20
Comparing two strings returned -1
```

Unicode strings are different subject and I won't explain it in details here. I hope you know it a bit. In case you don't, here is a quick introduction: Traditional string characters of type `char` are single-byte and encoded in so called ANSI code page. It means they contain ASCII characters and the meaning of upper 128 codes (or rather negative codes, as `char` in C and C++ is really a signed integer number) change meaning depending on the current system codepage. For example, in Poland we use Windows-1250 (CP1250) codepage so there are some values assigned to diacritic letters we use in our language, like ą, ć, ł, ż.

Unicode, on the other hand, is a standard that allows encoding characters from different languages in a single string, but requires more bytes per character. There are multiple ways of encoding Unicode strings into bytes, like UTF-8 widely used in the Web. But what we mean here by Unicode is the UTF-16 encoding, in which every character is a 16-bit unsigned number (takes two bytes). That's what `wchar_t` type does (at least in Windows, in Visual C++ compiler) and also the `std::wstring` class. To write an Unicode string in C++ source code, we have to precede it with `L` letter, like you can see in the example above. Such string is of type `const wchar_t*`.

Managed Strings

As I said before, .NET platform has its own, standard way of handling strings – the `System::String` class. They must be used by managed reference `^` and allocated with `gcnew`. Characters of these strings are of type `wchar_t` – 16-bit Unicode. You can initialize them with Unicode constants as there is an implicit conversion between `const wchar_t*` and `System::String`. Initializing managed strings with ANSI strings also works and the characters are automatically converted to Unicode.

Managed strings can be concatenated with `+` operator. You can also call its methods like `ToUpper`.

```
using System::String;
String ^managed_from_wide = gcnew String(native_wsz);
String ^managed_from_ansi = gcnew String(native_sz);
Console::WriteLine(L"managed_from_wide contains: " + managed_from_wide);
Console::WriteLine(L"managed_from_ansi contains: " + managed_from_ansi);
Console::WriteLine(L"ToUpper() returned: " + managed_from_wide->ToUpper());
```

The example above prints following output:

```
managed_from_wide contains: Native Unicode C string
managed_from_ansi contains: Native ANSI C string
ToUpper() returned: NATIVE UNICODE C STRING
```

String Conversions

If we have so many different ways of storing strings in C++/CLI, a question arises about how to convert between them. Best solutions is of course not to do it – it's always more efficient to keep data in the most natural format, the one that is finally needed. But sometimes conversion is necessary. Here is how I do it:

You've already seen a conversion from C string (`const char*` or `const wchar_t*`) to managed string (`System::String`). It's done by the `String` class constructor. To convert STL string (`std::string`) to managed string, I just retrieve its C string with `c_str()` method and then create managed string, like this:

```
std::string native_str = "Native ANSI C++ string";
String ^managed_from_stl = gcnew String(native_str.c_str());
```

Same effect can be achieved with `PtrToStringAnsi` method from `Marshal` class. The `System::Runtime::InteropServices::Marshal` static class contains many methods useful for interoperability between managed and native world (for pointers, data buffers, strings, errors, COM stuff, memory allocations and more) so I recommend taking a look at it.

```
String ^managed_from_stl = System::Runtime::InteropServices::Marshal::PtrToStringAnsi(
  (IntPtr)(void*)native_str.c_str());
```

Conversion in the opposite way is a little bit more complicated as it requires to somehow allocate a native buffer for characters. The way I do it in my code is using `Marshal::StringToHGlobalAnsi` method that allocates and fills a native memory buffer with null-terminated, ANSI string initialized from a managed string. I can then copy these data to desired place, like into an STL string, before I have to free the buffer with call to `Marshal::FreeHGlobal`. My function for converting managed to native strings is:

```
static void ClrStringToStdString(std::string &outStr, String ^str) {
  IntPtr ansiStr = System::Runtime::InteropServices::Marshal::StringToHGlobalAnsi(str);
  outStr = (const char*)ansiStr.ToPointer();
  System::Runtime::InteropServices::Marshal::FreeHGlobal(ansiStr);
}
```

Conversions to and from Numbers

Many programming languages treat atomic types (like `int`) a little bit like objects and allows calling methods on them. In .NET there are even class for it. Basic types like `int` or `double` are aliases to `System::Int32` and `System::Float`. Native C++ doesn't have such feature, but as C++/CLI is a .NET language, it does. C++/CLI has no separate types for native and managed `bools`, `ints`, `floats` etc. It means you can call methods on atomic types in C++/CLI, like very useful `ToString`:

```
int i = 1016;
String ^i_str = i.ToString(); // Method call on atomic type!
String ^message = L"The number is: " + i_str;
Console::WriteLine(message); // The number is: 1016
```

Conversion from string to number is done by static method `Parse` or `TryParse`, existing in all numeric types like `int`, `short`, `float`, `double` etc. Difference between them is that `TryParse` returns false in case of parsing error, while `Parse` throws an exception is such case. Example:

```
String ^i_str = gcnew String(L"1016");
int i = int::Parse(i_str); // Static method call on atomic type!
Console::WriteLine(L"Parsed number is: {0}", i);
// Parsed number is: 1016
```

Building Strings

Strings in .NET are immutable. It means that once created, a string cannot be changed. Other than native `std::string`, there are no methods in `System::String` for inserting or removing characters. Even single characters accessed with `[]` operator are read-only. To change a managed string, you have to create a new one and assign it to your `String^` variable. That's why `ToUpper` method returns a new string, it cannot make characters uppercase in place.

So, how do we construct some complex string, you may ask? There are many ways.

1. First is to concatenate parts with `+` operator into a final string.

```
unsigned year = 2011, month = 12, day = 24;
String ^date = year.ToString() + L"-" + month.ToString() + L"-" + day.ToString();
// date == L"2011-12-24"
```

2. It can be done more conveniently (and also more efficiently I believe) with `String::Format` static method, taking formatting string and variable number of arguments of any type. The formatting string is in special format used by .NET, with places for data referred by index in curly braces, like `{0}`. It's completely different from formatting strings in `printf` from C standard library, which use `%` sign.

```
unsigned year = 2011, month = 12, day = 24;
String ^date = String::Format(L"{0}-{1}-{2}", year, month, day);
// date == L"2011-12-24"
```

3. Because strings in .NET are immutable, there is a separate class for building and modifying strings called `StringBuilder`. Object of this type holds a mutable buffer of characters and the class defines a lot of methods to manipulate it. You can append and insert data of different types (automatically converted to string), as well as remove parts of the string. There is also `AppendFormat` method that takes formatting string and variable number of arguments, like `String::Format` method. Finally you call `ToString` and get a normal String object.

```
unsigned year = 2011, month = 12, day = 24;
System::Text::StringBuilder sb;
sb.Append(year);
sb.Append(L'-');
sb.Append(month);
sb.Append(L'-');
sb.Append(day);
String ^date = sb.ToString();
// date == L"2011-12-24"
```

Value Formatting

Sometimes default way of converting number to string is not enough. Then we can use special format strings. For example, `"D2"` means that a number should be shown as decimal and have at least two digits, padded with zeros if necessary. This is needed when formatting days and months of dates. Such format string can be passed as optional parameter to `ToString` method or appended in a bigger format string after a colon, e.g. when used in `String::Format` or `Console::WriteLine`. For example:

```
unsigned year = 2011, month = 1, day = 3;
String ^day_bad  = day.ToString();      // "3"
String ^day_good = day.ToString(L"D2"); // "03"
String ^date_bad  = String::Format(L"{0}-{1}-{2}",          year, month, day); // "2011-1-3"
String ^date_good = String::Format(L"{0:D4}-{1:D2}-{2:D2}", year, month, day); // "2011-01-03"
```

Other format string is hexadecimal form. If you want to show the value of a pointer or other 32-bit identifier, you would probably want to see it as 8-digit hexadecimal number with upper letters. This can be achieved with:

```
int variable;
int *pointer = &variable;
unsigned pointer_val = (unsigned)pointer;
String ^pointer_str = pointer_val.ToString(L"X8"); // "0012F380" on my machine
```

One more format string I'd like to show is precision of floating point numbers. It can be specified using format string like `"F2"`, where 2 is a number of digits after decimal mark. For example:

```
double pi = Math::PI;
Console::WriteLine(pi.ToString());      // 3,14159265358979
Console::WriteLine(pi.ToString(L"F2")); // 3,14
```

Decimal Mark Issues

As you see in the last example, my system converted floating-point number to string using comma as decimal mark. If you expected period instead, you have to know this depends on the current system locale. English-speaking countries use period as decimal mark (and comma to separate thousands), while we in Poland, like in other European countries, use comma as decimal mark. That's a real issue in programming – believe me, I've seen commercial software that didn't work because it was expecting period while operating system provided numbers with comma.

.NET framework applies current locale to all conversions between strings and numbers. That's not different than what other API-s do. Event functions from standard C library like `gcvt` or `printf` produce comma as decimal mark, if `setlocale(LC_ALL, "polish");` is called before. But .NET automatically selects locale for you so if you want to overcome this problem, you have to actively request your conversion to be "culture invariant", thus default to period as decimal mark. To parse string as floating-point number using period as decimal mark:

```
String ^pi_str = L"3.14";
double my_pi = double::Parse(
  pi_str,
  System::Globalization::NumberStyles::Float,
  System::Globalization::CultureInfo::InvariantCulture);
// my_pi == 3.14
```

To convert floating-point number to string using invariant culture, pass the appropriate static culture object to the `ToString` method:

```
double pi = Math::PI;
Console::WriteLine(pi.ToString(
  System::Globalization::CultureInfo::InvariantCulture)); // 3.14159265358979
```

Locale can also be set globally for the current thread like in the code below. Pay attention however that locale currently set for .NET libraries does not affect the native standard C/C++ library and vice versa. They can be different.

```
System::Threading::Thread::CurrentThread->CurrentCulture =
  System::Globalization::CultureInfo::InvariantCulture;
```

Enumerations

Just like with classes, C++/CLI also differentiates native and managed enums. Their syntax is very different because former follow rules of native C++, while latter follow rules that apply to enums in C# and the whole .NET. Native enums are defined using `enum` keyword, while managed enums start with `enum class` keyword. For example:

```
enum NativeMood {
  NativeHappy, NativeSad
};
enum class ManagedMood {
```

```
  ManagedHappy, ManagedSad
};
```

These two enums don't look very different right now, but as you will see, they have to be used differently. Of course we can use both types to define variables and store them by value. All in all enum is just a number and the rest is the matter of language syntax. But to use one of the values defined inside, native enum must be qualified with enum name followed by `::` and then value name, while native enum should not be qualified as enums in C++ do not form scope, values from inside pollute global namespace. Let's see the example:

```
// Native enum should be unqualified.
NativeMood native_mood;
// OK
native_mood = NativeHappy;
// warning C4482: nonstandard extension used: enum 'NativeMood' used in qualified name
native_mood = NativeMood::NativeHappy;

// Managed enum should be qualified.
ManagedMood managed_mood;
// error C2065: 'ManagedSad' : undeclared identifier
managed_mood = ManagedSad;
// OK
managed_mood = ManagedMood::ManagedSad;
```

Another difference is that value from native enum can be used as integer number because these types can be implicitly converted. With managed enum it's not the case – compiler prints error about types incompatibility and explicit type cast must be used.

```
unsigned u;
// OK
u = native_mood;
// error C2440: '=' : cannot convert from 'ManagedMood' to 'unsigned int'
u = managed_mood;
// OK
u = (unsigned)managed_mood;
```

Type Casts

Casting between data types in native C++ can be done using old C notation `(Type)Value` or new C++ operators: `static_cast<Type>(Value)`, `reinterpret_cast<Type>(Value)`, `const_cast<Type>(Value)`, `dynamic_cast<Type>(Value)`. This is all available in C++/CLI too and works as you may expect. But C++/CLI also defines a new casting operator – `safe_cast<Type>(Value)`. It works very intuitively, can be applied to convert between many different kinds of types and it performs runtime checks so it guarantees safety. As we have lots of types available in C++/CLI language – atomic types, native classes, managed classes etc. – I won't cover all possible cases here, but here are some more interesting examples involving managed types:

```
ref class BaseClass { };
ref class DerivedClass1 : public BaseClass { };
ref class DerivedClass2 : public BaseClass { };

int main(array<System::String ^> ^args)
{
  DerivedClass1 ^derived_object_1 = gcnew DerivedClass1();
  // Upcast does not require explicit cast.
  BaseClass ^base_object = derived_object_1;
  // base_object is here really of type DerivedClass1, not DerivedClass2, so...

  DerivedClass2 ^derived_object_2;
```

```
  // Invalid downcast with safe_cast throws InvalidCastException with message:
  // "Unable to cast object of type 'DerivedClass1' to type 'DerivedClass2'."
  derived_object_2 = safe_cast<DerivedClass2^>(base_object);
  // Invalid downcast with dynamic_cast returns nullptr.
  derived_object_2 = dynamic_cast<DerivedClass2^>(base_object);
```

Boxing

Additional subject connected with type casts is boxing. Boxing is a feature of .NET platform where a value of some type that should be stored by value, not by managed reference (atomic types like `int` or C# structs, defined in C++/CLI as `value class` or `value struct`) is saved into managed heap. It's especially useful if we want to keep such value or collection of such values in a place where `Object^` type is expected. `Object` is a base class for all other types in .NET. Example place where any user-defined object can be stored is `Tag` property of every Windows Forms control. So for example if you want to associate integer number with a button on your form, you can use boxing and unboxing in C++/CLI like this:

```
int number = 123;
// Boxing. Cast is not required.
button1->Tag = number;
...
// Unboxing. Use safe_cast operator.
int recovered_number = safe_cast<int>(button1->Tag);
// recovered_number == 123
```

Properties

Property is a class member that can be used like a field (member variable) – assigned and retrieved, while inside it executes some specific code to write or read its value, like there were getter and setter methods. It is another programming language feature that most modern, high-level programming languages (including .NET) have while native C++ is lacking. Obviously it's not necessary, although convenient.

To be compatible with .NET, C++/CLI extends the C++ syntax to support properties in classes. They are defined using `property` keyword. A property must have methods inside called `get` and `set` that use appropriate types (unless you want to create a write-only or read-only property, which is also correct).

```
ref class ManagedMonster {
public:
  ManagedMonster(int HP) : m_HP(HP) { }
  property int HP {
    int get() { return m_HP; }
    void set(int val) { m_HP = val; }
  }
private:
  int m_HP;
};
```

This property just writes and reads a private field, but any code can be execute in its getter and setter. For example, setter can trigger some additional actions connected with changing the parameter, while getter can calculate final value instead of reading it directly from a field. Regardless of this however, usage of a property looks like direct access to a class field:

```
ManagedMonster ^monster = gcnew ManagedMonster(100);
// HP set is called.
monster->HP = 200;
// HP get is called. Prints 200.
Console::WriteLine(monster->HP);
```

Such properties can be defined only in managed classes, not inside natives ones.

Exceptions

Both native C++ and .NET support error handling via the concept of exceptions. Of course they differ in details. In practice of native C++ programming exceptions are often not used at all and when they are, there is no widespread standard of what to throw. Language allows throwing and catching values of any type, like ints, bools, objects of user-defined classes or pointers. .NET platform, on the other hand, establishes a standard that only objects of class `Exception` or derived classes can be thrown as exceptions.

Good news is that both native and managed exceptions can be handled in C++/CLI using same syntax and freely mixed, even inside single `try…catch` block. Let's see an example.

```
using namespace System;

#include <exception> // for native std::exception

void NativeErroneousFunction() {
  throw std::exception("A native error!");
}

void ManagedErroneousFunction() {
  throw gcnew System::Exception(L"A managed error!");
}

int main(array<System::String ^> ^args) {
  try {
    //NativeErroneousFunction();
    //ManagedErroneousFunction();
  }
  catch (const std::exception &ex) {
    Console::WriteLine(L"Native error! " + gcnew String(ex.what()));
  }
  catch (System::Exception ^ex) {
    Console::WriteLine(L"Managed error! " + ex->Message);
  }
  finally {
    Console::WriteLine(L"Finalization.");
  }

  return 0;
}
```

Two types of exceptions are used here. `NativeErroneousFunction` throws a native exception of type `std::exception` – a class defined in standard C++ library, in `<exception>` header, intended to be used or subclassed to handle errors in native C++ (who uses it? :) Object of this class keeps pointer to a externally-owned null-terminated string with error message, which can be retrieved using `what` method. We throw this exception by value. Second type of exception is an object of managed class `System::Exception`, allocated on the heap with `gcnew` operator. It holds managed string with error message, accessed through `Message` property. By uncommenting call to first or second "erroneous" function, you can test throwing first or second kind of exception.

Exceptions are catched in `try…catch` block, which has common syntax for native as well as managed code. As you can see, both types of exceptions are caught here – native one by a reference to const object and managed one by a managed pointer `^`.

There is also `finally` section – another very useful feature that most modern programming languages have, while native C++ does not. I assume you already know how it works. As it is also used in .NET platform, Microsoft added support for `finally` to C++/CLI so we can use it in `try` section, no matter if we intend to handle native or managed exceptions. A `try` can have only `finally` section, only `catch` sections or both.

There is one more thing you have to know about the order of handling different types of exceptions in `catch` blocks. As you know from native C++, all types of exceptions can be caught using `catch (...)`. In .NET, on the other hand, all exceptions are subclasses of `System::Exception` class, so `catch (Exception^)` will handle all possible cases. Question is how these general classes relate to each other? Surprisingly all native exceptions, no matter what type they have, are also handled by `catch (Exception^)`, because a native exception is implicitly converted to an object of type `System::Runtime::InteropServices::SEHException` with message "External component has thrown an exception.". That's why the `catch` section that handle `System::Exception` type, if exists, is absolutely most general so it must be the last `catch` section in the block, even relative to any native exception types – like in the example above.

Arrays

Array is undoubtedly the simplest and most fundamental type of container. Both C++ and .NET support arrays, so we also have them in C++/CLI, with clear distinction between native and managed ones – just like it was for classes.

Native Arrays

Native arrays look like in standard C++. You can define and use a static array like this:

```
int native_array_static[10];
for (size_t i = 0; i < _countof(native_array_static); ++i)
  native_array_static[i] = i;
// native_array_static: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Or you can dynamically allocate arrays on the native heap with operator `new[]`. Of course you must not forget to free them with operator `delete[]` when no longer needed.

```
size_t count = 10;
int *native_array_dynamic = new int[count];
for (size_t i = 0; i < count; ++i)
  native_array_dynamic[i] = i;
// native_array_dynamic: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
delete [] native_array_dynamic;
```

Exceeding range of a native array causes undefined behavior, so you can end up with program crash with access violation or an error posted by some runtime check telling about the corruption of the heap or stack (in best case), reading some random data or overwriting some other variables.

Managed Arrays

Array in .NET is a built-in type that must be allocated on the managed heap, has some properties and methods, including `Length` that returns the number of elements. In C# we use `[]` to define arrays, but in C++/CLI the syntax for managed arrays is very different. It uses `array` keyword that behaves like a class template. Let's look at the example:

```
array<int> ^managed_array = gcnew array<int>(10);
for (int i = 0; i < managed_array->Length; ++i)
  managed_array[i] = i;
// managed_array: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Of course managed array don't need to be freed manually as garbage collector will release its memory when the array is no longer referenced from any place in your code.

Exceeding range of a managed array throws an exception of type `System::IndexOutOfRangeException`.

`foreach` loop – a construct present in many programming language but not in C++ - is also available in C++/CLI in form of a `for each` keyword (it is so called whitespace keyword, Microsoft patented it!). With it you can iterate through many different kinds of containers, including managed arrays. For example:

```
for each (int i in managed_array)
  Console::Write(L"{0}, ", i);
// Output: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

Another nice language feature is array initialization during construction. C++ supports this for static arrays only, not for dynamically allocated ones. Fortunately we can do it in C++/CLI during construction of managed arrays. Here is a bunch of examples:

```
// OK
int native_array_static_1[10] = {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
};
// Also OK
int native_array_static_2[] = {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
};

// ERROR! There is no way to do it :(
int *native_array_dynamic = new int[10] {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
};

// OK
array<int> ^managed_array_1 = gcnew array<int>(10) {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
};
// Also OK
array<int> ^managed_array_2 = gcnew array<int> {
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
};
```

Containers

Every high-level programming language, to allow coding complex programs, needs to provide some containers that hold a collection of objects. C++ standard library – STL – has some template classes for that, like `std::vector`, `std::list`, `std::set`, `std::map`, `std::stack` etc. Microsoft added in Visual C++ some additional containers that are very useful but not defined by STL standard, like `stdext::hash_map`. You can use all of them in C++/CLI to hold native objects, like this:

```
#include <map>
#include <string>
#include <iostream>

...

typedef std::map<std::string, NativeMonster*> MonsterMap;
MonsterMap native_map;

native_map.insert(MonsterMap::value_type("Goblin", new NativeMonster(10)));
native_map.insert(MonsterMap::value_type("Boss", new NativeMonster(1000)));

for (MonsterMap::iterator it = native_map.begin(); it != native_map.end(); ++it) {
```

```
  std::cout << "Monster: " << it->first << std::endl;
  it->second->TellHitPoints();
}

for (MonsterMap::iterator it = native_map.begin(); it != native_map.end(); ++it)
  delete it->second;
```

This program prints following output:

```
NativeMonster Constructor
NativeMonster Constructor
Monster: Boss
NativeMonster has 1000 HP
Monster: Goblin
NativeMonster has 10 HP
NativeMonster Destructor
NativeMonster Destructor
```

.NET platform has many container classes in its standard library. Old ones, which hold objects of type `Object^` so they need casting or boxing your real data type – are in `System::Collecions` namespace. New ones – generic classes parameterized with your real data type – are in `System::Collections::Generic` namespace. C++/CLI supports defining as well as using generics – a .NET equivalent of C++ templates – but I won't describe it in this tutorial. Instead, let's just see how we can use a managed container to hold object of a managed class.

```
typedef System::Collections::Generic::Dictionary<String^, ManagedMonster^> DictionaryType;
typedef System::Collections::Generic::KeyValuePair<String^, ManagedMonster^> KeyValuePairType;
DictionaryType ^managed_dictionary = gcnew DictionaryType();

managed_dictionary->Add(L"Goblin", gcnew ManagedMonster(10));
managed_dictionary->Add(L"Boss", gcnew ManagedMonster(1000));

for each (KeyValuePairType it in managed_dictionary) {
  Console::WriteLine(L"Monster: {0}", it.Key);
  it.Value->TellHitPoints();
}
```

This program prints following output:

```
ManagedMonster Constructor
ManagedMonster Constructor
Monster: Goblin
ManagedMonster has 10 HP
Monster: Boss
ManagedMonster has 1000 HP
```

Locking

Many things exist in .NET platform to support parallel (multithreaded) programming. Probably the simplest way of synchronizing access to an object so it is mutually exclusive between threads is locking. You can just lock any object forming a critical section around some code, so then you can be sure that no more than one thread executes this code at time. C# has a language construct for that: `lock (object) { code }`. In C++/CLI you can achieve the same by including `<msclr\lock.h>` header and using an `msclr::lock` class. It is RAII – it creates lock in constructor and frees it in destructor, so it is best used as a local object inside some scope, created on the stack – like this:

```
#include <msclr\lock.h>
...
Object ^m_SyncObj;
...
```

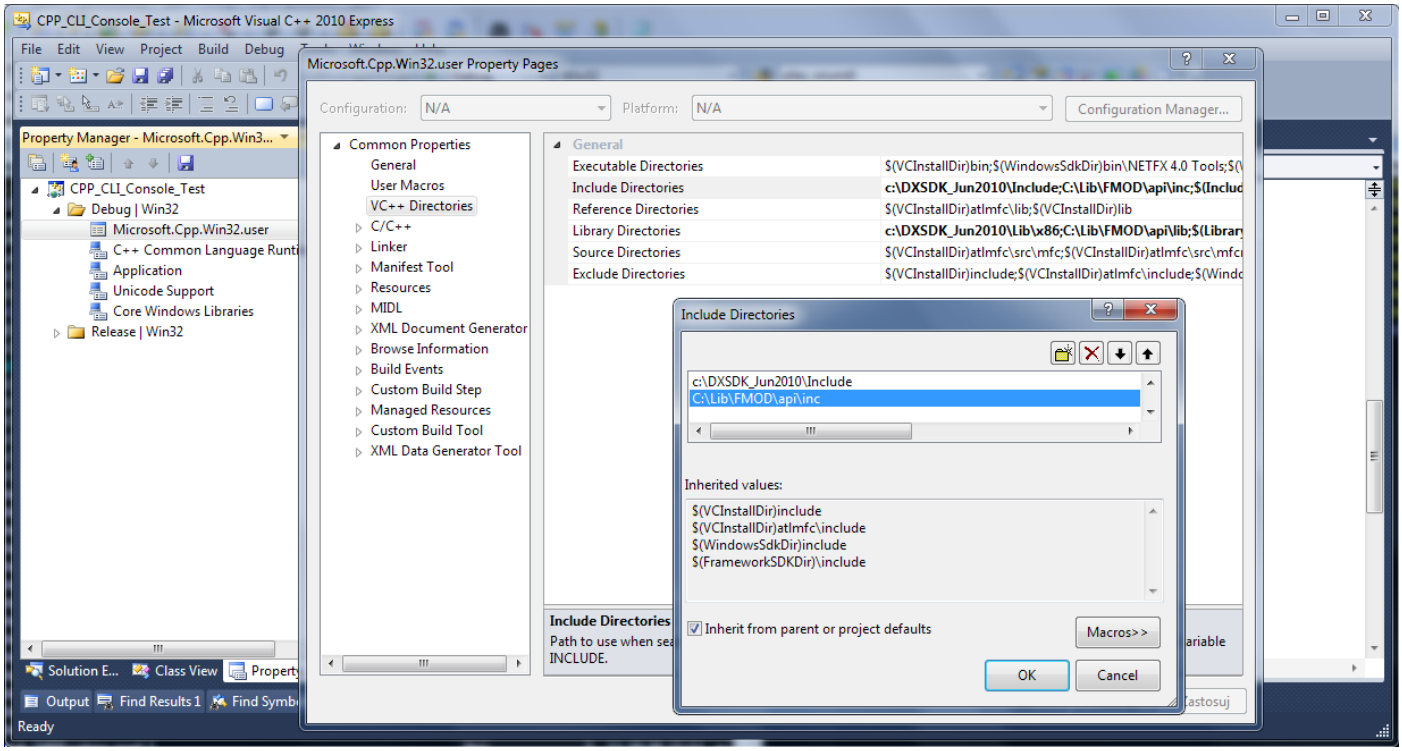
```
{ // A scope for critical section
  msclr::lock(m_SyncObj);
  ... // Critical section code
}
```

Using Libraries

One of the biggest strengths of C++/CLI is that you can not only freely mix managed and native code in your project, but also directly use native C and C++ libraries next to .NET libraries. Way of referencing them is of course different, natural to the particular type.

Using Native Libraries

To prepare for using a native C or C++ library, you have to follow same steps as if you coded a native C++ program. First, open the `Property Manager` panel. Then select `Microsoft.Cpp.Win32.user` property sheet, right-click on it and select `Properties`. In the `Property Pages` dialog window select `VC++ Directories` item in the tree on the left. Then from the list on the right edit `Include Directories` and then `Library Directories` item, adding appropriate paths to `h` and `lib` files of the library, respectively.



To use such library in your project, you also have to fulfill all that is required to use a library in native C++. First, include necessary header in your `cpp` file, like this:

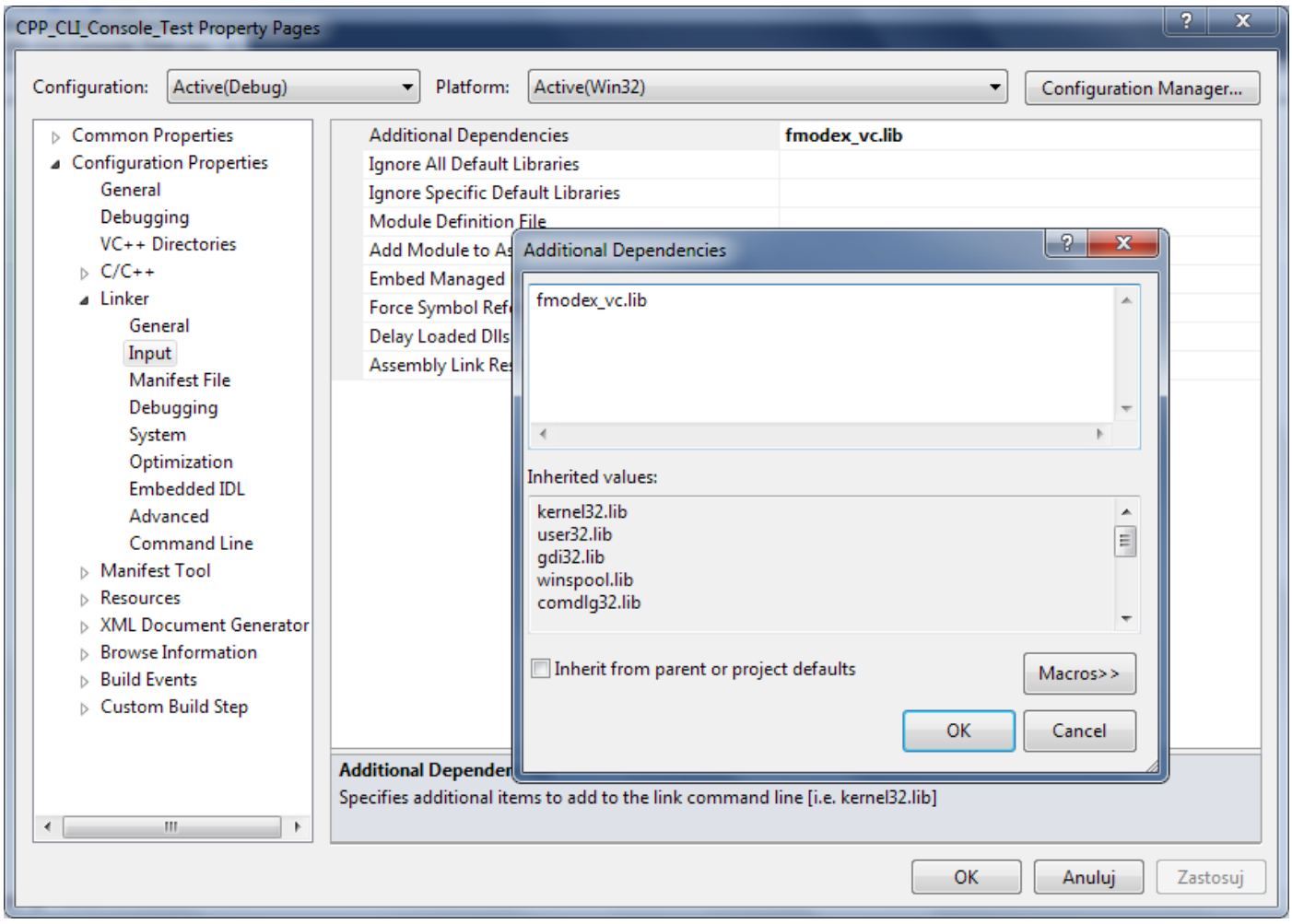
```
#include <fmod.h>
```

This allows you to compile a code that uses the library without errors. But to be also able to link your program successfully and build the final `exe` file, without “unresolved external symbol” linker errors like this:

```
error LNK2019: unresolved external symbol "extern "C" enum FMOD_RESULT __stdcall
FMOD_System_Create(struct FMOD_SYSTEM * *)"
(?FMOD_System_Create@@$$J14YG?AW4FMOD_RESULT@@PAPAUFMOD_SYSTEM@@@Z) referenced in function "int __clrcall
main(cli::array<class System::String ^ >^)" (?main@@$$HYMHP$01AP$AAVString@System@@@Z)
```

You also have to tell the linker to link with a `lib` file corresponding to the header used. You can do it in two ways. First is to open project properties, navigate to `Configuration Properties / Linker / Input` and add names of

required `lib` files to the `Additional Dependencies` item. Just don't forget to do it for both `Debug` and `Release` configurations!

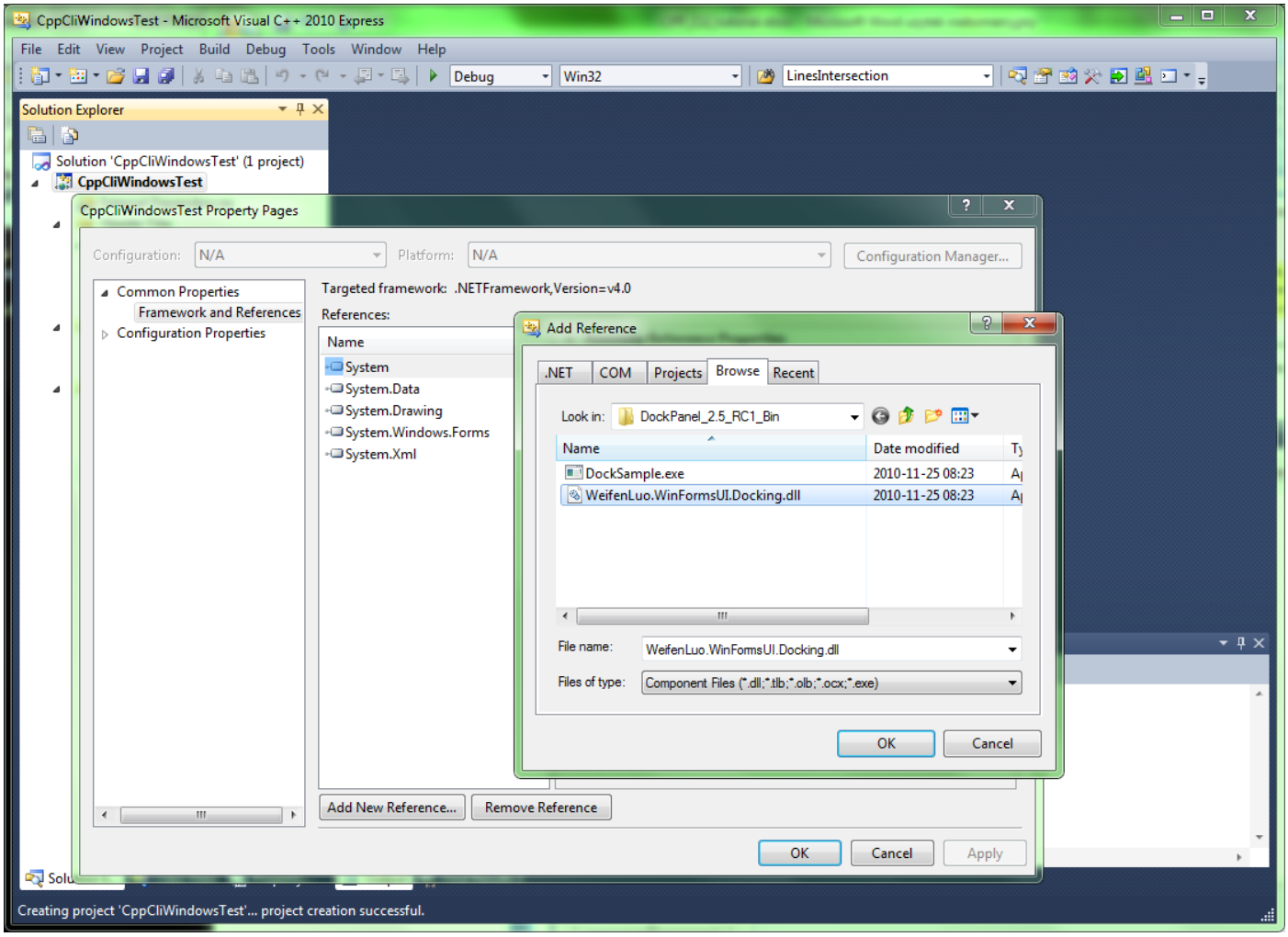


Alternative way is to put a special, Microsoft-specific directive anywhere in your project code:

```
#pragma comment(lib, "fmodex_vc.lib")
```

Using Managed Libraries

To reference external managed library – a one written in C# or any other .NET language – also open project properties, but then select `Common Properties / Framework and References` in the tree on the left. Then click `Add New Reference` button, in the new `Add Reference` dialog window select `Browse` tab, navigate to the directory where the library is located and select appropriate `dll` file.



Now you can start use classes from that library. Different than in native C++, there is no need for additional including or linking.

This reference list in project properties is also useful for other purposes. From there you can click `Add New Reference` button again, but in the `Add Reference` window go to the `.NET` tab this time. On the list that appears you can select and add to your project references additional components of .NET library that are not referenced by default, like `System.Windows.Forms.DataVisualization`, which contains a very powerful control for displaying charts, introduced in .NET version 4.

Same way you can establish dependencies between projects in your Visual C++ solution. You just need to switch to the `Projects` tab and select some project that the project you currently edit should depend on.

Native Types in Exported Functions

There is one issue with coding libraries in C++/CLI that appears when you export some functions that use external, native types. I will explain in step-by-step using an example. Imagine you want to code a separate library that implements a logger to be able to print different types of messages on the system console. The library is written in C++/CLI and its header file looks like this:

```
#pragma once

#include <exception>
#include <string>

using namespace System;
```

```
namespace Library {
  public ref class Logger {
  public:
    void PrintManagedString(String ^s) {
      Console::WriteLine(s);
    }
    void PrintNativeString(const char *s) {
      Console::WriteLine(gcnew String(s));
    }
    void PrintNativeException(const std::exception &ex) {
      Console::WriteLine(L"Error: " + gcnew String(ex.what()));
    }
    void PrintStlString(const std::string &s) {
      Console::WriteLine(gcnew String(s.c_str()));
    }
  };
}
```

As you can see there are four printing methods. Should you be able to reference such library in another C++/CLI project, instantiate `Library::Logger` class and use these methods without any problems? It turns out that it is not the case. A code that uses first two methods compile successfully and work as you could expect, because `PrintManagedString` uses managed type (`System::String`) as parameter and `PrintNativeString` uses atomic type (`char`).

But when you try to use third method, a compilation error appears:

```
error C3767: 'Library::Logger::PrintNativeException': candidate function(s) not accessible
```

It means that the native type used as parameter for the `PrintNativeException` method (`std::exception`) is considered “private” in .NET world, which makes this method inaccessible from outside of your library. Luckily this can be fixed with a special pragma directive that makes specified type public (`std::exception` in our case), which also makes methods that use this type as parameter or return value type accessible for users of your library. All you need to do is to put following line somewhere in your header file:

```
#pragma make_public(std::exception)
```

But that’s not all I can say in this topic. I have a bad news regarding fourth method from the `Logger` class we discuss here. You can’t do the same `#pragma make_public` trick with `std::string` type. If you try it, compiler will tell you that you cannot do that with template types, while `std::string` is really a template – a `typedef` to `std::basic_string<char>`. There is no workaround for this unfortunately. You have to cast such objects to `void*` or something…

```
error C2158: 'std::basic_string<_Elem,_Traits,_Ax>' : #pragma make_public directive is currently
supported for native non-template types only
```

Summary

That’s all I wanted to tell you about C++/CLI programming language. I hope you enjoyed the tutorial, understood most of it and learned something that you can now use in your personal or professional projects. As you can hopefully see now, the exotic C++/CLI language is not so weird, unpleasant or useless as some people think. It is another language of .NET platform that has unique feature – with it you can freely mix native and managed code, which makes it a perfect choice for some types of programming projects. If you knew C++ and some .NET language like C# before, you only need to learn some syntax rules and your are ready to code in C++/CLI.

Obviously this document is not exhaustive. I tried to explain basics of this language based on my experience. I don't know everything about it and I didn't even written everything I know. For example, I didn't mention about the `<msclr/marshal.h>` header containing very useful utilities. I recommend you now, instead of Googling for another tutorial, look into official documentation for any further details. Select `Help / Manage Help Settings` in your Visual C++, click `Install Content from Online`, select and download some of the packages. This way you will have full .NET documentation offline, working fast and accessible whenever you need it. Just click `Help / View Help` or press `Ctrl+F1`. I recommend you keep it open all the time as you code and use `Intex` tab for navigation.