Programming the .NET Framework 4.5

Module 07 - Interoperability

In This Chapter

- Platform Invoke
- COM Interop
- C++/CLI
- Overview of CLR Hosting

Overview of Interoperability

- .NET was **designed for interoperability**

- Challenges Faced:
  - **Locating** the other side
  - Passing data correctly (**marshaling**)
  - **Lifetime** management (GC vs. …)

Platform Invoke

- **Managed** code calls **into native** DLL
- Native code calls into managed code

- Very **simple**
- **C-style** exported functions only
- Partial control over **marshaling**

COM Interoperability

- **Managed** code calls **into COM** component
- **Native** code calls **into managed** code as if it were a COM component

- **Standard** COM interfaces
- **Very limited** control over **marshaling**
- **Locating COM** components

C++/CLI

- **Managed** code calls **into C or C++** code
- **C or C++** code calls **into managed** code
- The most **powerful** .NET language

- **Full control** over **marshaling**
- **Fine-grained choice** of "what's managed"
- **Mixed** managed and native assemblies
- Significantly more **complicated**

P/Invoke

- Managed code calling native code
  - Custom **C-style** DLL
  - Windows API (Win32)

```
//Native signature:
   int IsPrime(int number);

 //Managed signature:
   [DllImport("MyDll.dll")]
   static extern bool IsPrime(int number);
```

Behind the Scenes

- `extern` satisfies the compiler

- `[DllImport]` satisfies the runtime
  - The **DLL** is **located**
  - The **entry point** is located
  - **Parameters** are converted
  - **Return** values are converted back

Basic P/Invoke

Demo

Marshaling

- Marshaling: **translating** parameters
- ***Blittable*** types have the same representation
  - `System.Int32` and C `int`
  - `System.Single` and C `float`
- Some types **do not!**
  - Strings, booleans, arrays vs. pointers

Standard Mappings

- Standard mappings are performed by P/Invoke
  - byte → unsigned char
  - long → __int64
  - bool → int
  - string → char* *or* wchar_t*

Character Mapping

- Can be performed on per-function basis

```
//Native signature:
   void wputs(wchar_t* s);

//Managed signature:
   [DllImport("MyDll.dll", CharSet=CharSet.Unicode)]
   static extern void wputs(string s);
```

Marshaling Individual Parameters

- Customization with [MarshalAs]

```
//Native signature:
    BOOL IsValid(LPCWSTR lpszText);

//Managed signature:
   [DllImport("MyDll.dll")]
   [return: MarshalAs(UnmanagedType.Bool)]
   static extern bool IsValid(
      [MarshalAs(UnmanagedType.LPWStr)] string text);
```

Calling the Windows APIs

- Most **Win32 APIs** have **two** versions:
  - `CreateFileA` – ANSI string
  - `CreateFileW` – Unicode string

Use `CharSet.Auto` with these functions!

Passing Structures

- C# structs will work
- Pointers are either:
  - Arrays, -or-
  - ref / out parameters

Structures and Pointers

```
//Native signature:
    void Find(CONST ITEM* items, DWORD count,
            ITEM* lookup, DWORD* index);

//Managed signature:
   [DllImport("MyDll.dll")]
   static extern void Find(
         ITEM[] items, int count,
         ref ITEM lookup, out int index);
```

Combining Unsafe Code

- Alternatively, pointers can be used
  - In unsafe context, requiring fixed

```
unsafe
{
        fixed(ITEM* p = &items[2])
        fixed(ITEM* q = &lookup)
        {
                Find(p, items.Length-2,
                     q, out index);
        }
}
```

Marshaling Mutable Strings

- If a string requires modification, use **`StringBuilder`** with a capacity

```
//Native signature:
    void FillString(char* s, char fill);

//Managed signature:
   [DllImport("MyDll.dll")]
   static extern void FillString(
       StringBuilder text, char fill);
```

Marshaling Structures and Strings

Demo

Marshaling Delegates

- Native-to-managed **callbacks**: delegates

```
//Native signature:
    typedef BOOL (__stdcall *PFNMATCH)(char* text);
    DWORD Find(PFNMATCH pfnMatch);

//Managed signature:
   delegate bool IsMatch(string text);
   [DllImport("MyDll.dll")]
   static extern int Find(IsMatch match);
```

Using Reverse P/Invoke

Passing a delegate (anonymous method):

```
IsMatch match = delegate(string s) {
        return s.StartsWith("A");
};
int index = Find(match);
```

Passing a delegate (lambda):

```
IsMatch match = s => s.StartsWith("A");
int index = Find(match);
```

Marshaling Delegates – Caution

- Keep the delegate alive until it's unused:
  - Store a static/member reference
  - Use `GC.KeepAlive`
  - Use `GCHandle`

Losing a Delegate

Demo

Generating Signatures

- Remembering signatures is tedious!
- **PInvoke.net** – a collection of signatures online
- **P/Invoke Interop Assistant** – a tool for generating signatures

Enumerate Window

Lab

P/Invoke Summary

- Easy automatic location and marshaling
- Customizing marshaling for strings, arrays, structures and pointers
- Works with C-style exported DLL functions only

COM Interoperability

- Managed code **calling COM** objects
- Native code calling .NET components exposed as COM objects
- Interoperability enabled by **runtime wrappers**

COM Interoperability - Challenges

- COM defines more rules than C
  - **Error handling** (`HRESULT, IErrorInfo`)
  - **Life-time** management (reference counting)
  - **Threading** model (apartments)
- And of course, there's marshaling!

COM Objects From Visual Studio

- Project → Add Reference → COM
- An *interop assembly* is generated

- Alternative: Use the *tlbimp.exe* tool

Non-Standard Mappings

- Sometimes the **generated interop** assembly is **wrong**!

- The only way to customize marshaling is by **manually editing** the interop assembly

Manual Customization

- Disassemble:

```
ildasm Interop.MyCOM.dll /out:Interop.MyCOM.il
```

- Perform modifications, e.g.:
  - int32& → int32[] marshal([])

- Reassemble:

```
ilasm /dll Interop.MyCOM.il /resource:Interop.MyCOM.res
```

Primary Interop Assemblies

- Signed interop assembly supplied by the **component's vendor**
  - Marked with `[PrimaryInteropAssembly]` attribute
- Create with *tlbimp.exe* ***/primary***

Reflection and IDispatch

- What if there's no type library?
- What if I'm referencing COM objects **dynamically**?

- Use Reflection – it will use **IDispatch**!
  - `Type.GetTypeFromProgID`
  - `Activator.CreateInstance`
  - `Type.InvokeMember`

Lifetime Management

- .NET objects are subject to **GC**
- COM objects are subject to **RC**

- When the RCW is collected, it releases the COM object
  - `Marshal.ReleaseComObject`
  - `Marshal.FinalReleaseComObject`
- **Beware of RCW-CCW cycles!**

Error Handling

- Failure **HRESULTs** are converted to **CLR exceptions**
- Success HRESULTs are not mapped

Threading Models

- COM objects can specify an apartment requirement
- A .NET thread has a default apartment mode of MTA
  - `Thread.ApartmentState`
  - `[STAThread], [MTAThread]`

Accessing COM Objects from .NET

Demo

.NET Objects as COM Components

- .NET objects **exposed as COM** components can be accessed from almost any Windows language!
- A type library is **generated** from the assembly
  - Registered under HKCR
- The .NET assembly is not in the registry!

Visual Studio Integration

- Project → Properties → Build → Register for COM Interop
- In *AssemblyInfo.cs*, use `[assembly:ComVisible(true)]` ***-or-***
- Specify `[ComVisible(true)]` for individual types, interfaces, methods etc.

C++ Clients

- Clients can use the type library as usual:

```
#import "MyNetObject.tlb" no_namespace

IMyNetClassPtr p(CLSID_MyNetClass);
p->MyMethod();
p->MyProperty = 5;
```

Alternatives for Registration

- regasm
- tlbexp

Exposing Interfaces

- Expose explicit interfaces
- `[ClassInterface]`

Limitations

- Customizing **marshaling is impossible**
- **Static members** can't be exposed
- **Overloaded** methods can't be exposed
- **Exceptions** are reported as HRESULTs

Accessing .NET Types from COM

Demo

Dynamic Dispatch
Regex Wrapper

Lab

COM Interoperability Summary

- Simple marshaling model
  - Hardly any or no customizations
- Overcoming the differences:
  - Registration model
  - Lifetime management
  - Error handling
  - Threading model

C++/CLI: The Most Powerful .NET Language

- Deterministic Cleanup
- Templates
- Native Types
- Multiple Inheritance
- STL
- Generic Algorithm
- Pointers
- Copy Constructor
- Assignment Operator
- Legacy Code

VS.

- GC, Finalizer
- Generics
- Reference & Value Types
- Interfaces
- Safe, Verifiable
- Security
- Properties
- Delegates, Events
- .NET Framework

What Is C++ On The CLR?

- ISO standard **C++** on the **CLR**
- Language binding to .NET framework
- Seamless **managed** and **native** interop

C++ Code Generation

- Compiles to MSIL
  - **/CLR** – Mixed Mode Images (native and MSIL)
  - /CLR:Pure – MSIL Only
  - /CLR:Safe – Verifiable MSIL
  - /CLR:oldSyntax – Managed C++
- Compiles to Native

Basic Class Declaration Syntax

- Types are declared “adjective class”:
- Fundamental types are mapped to each other

```
class CNative
ref class CManaged
value class CValue
interface class IInterface
enum class E
```

More Class Declaration Examples

- **Extending** the ISO C++ language:
- The same syntax for native and managed types

```
class CShape abstract ...
class CNative sealed ...
class CDerived : public CNative {};
   //Error: CNative is sealed
```

Declaring Properties

```
 1 value class Complex
 2 {
 3 public:
 4     property double Real;
 5     property double Imaginary;
 6     property double R
 7     {
 8         double get();
 9         void set(double newR);
10     }
11     property double Theta
12     {
13         double get();
14         void set(double newTheta);
15     }
16 };
```

Implementing Properties

```
1 double Complex::R::get() {
2     return Math::Sqrt(Real*Real + Imaginary*Imaginary);
3 }
4 void Complex::R::set(double newR) {
5     Real = Math::Cos(Theta) * newR;
6     Imaginary = Math::Sin(Theta) * newR;
7 }
8 double Complex::Theta::get() {
9     return Math::Atan2(Imaginary, Real);
10 }
11 void Complex::Theta::set(double newTheta) {
12     Real = Math::Cos(newTheta) * Real;
13     Imaginary = Math::Sin(newTheta) * Real;
14 }
```

Using Properties

```
1 Complex complex(2.0, 3.0);
2 Console::Write("{0} + {1}i",
3     complex.Real, complex.Imaginary);
4 Console::WriteLine(
5     " = {0:F2}*(cos({1:F2}) + i*sin({1:F2}))",
6     complex.R, complex.Theta);

                                    //Output:

  2 + 3i = 3.61*(cos(0.98) + i*sin(0.98))
```

Delegates and Events

```
 1 ref class ChatClient
 2 {
 3 private:
 4     void OnMessageArrived(
 5         System::Object^ sender,
 6         MessageArrivedEventArgs^ args);
 7     System::String^ _name;
 8     ChatServer^ _server;
 9 public:
10     ChatClient(System::String^ name,
11                ChatServer^ server);
12     void SendMessage(System::String^ message);
13 };
```

Delegates and Events (contd.)

```
 1 delegate void MessageArrivedEventHandler(
 2         System::Object^ sender,
 3         MessageArrivedEventArgs^ e);
 4
 5 ref class ChatServer
 6 {
 7 public:
 8     void SendMessage(
 9         System::String^ from,
10         System::String^ message);
11     event MessageArrivedEventHandler^ MessageArrived;
12 };
```

Delegates and Events (contd.)

```
 1 void ChatServer::SendMessage(...)
 2 {
 3     MessageArrived(this,
 4         gcnew MessageArrivedEventArgs(from, message));
 5 }
 6 ChatClient::ChatClient(...)
 7     : _name(name), _server(server)
 8 {
 9     _server->MessageArrived +=
10         gcnew MessageArrivedEventHandler(this,
11             &ChatClient::OnMessageArrived);
12 }
13 void ChatClient::SendMessage(System::String^ message)
14 {
15     _server->SendMessage(_name, message);
16 }
```

Virtual Functions

```
1 interface class I1 { int f(); int h(); };
2 interface class I2 { int f(); int i(); };
3 interface class I3 { int i(); int j(); };
4
5 ref class R : I1, I2, I3 {
6 public:
7 virtual int e() override; //error, no virtual e()
8 virtual int f() new; //new slot, doesn't override
   any f
9 virtual int f() sealed; //overrides I1::f and I2::f
10 virtual int g() abstract; //same as "=0“
11 virtual int x() = I1::h; //overrides I1::h
12 virtual int y() = I2::i; //overrides I2::I
13 virtual int z() = j, I3::i //overrides I3::j and
   I3::I
14 };
```

Explicit, multiple and renamed overriding

Fundamental Language Constructs

Demo

% IS TO ^

AS

& IS TO *

Storage And Pointer Model

- On the native heap (native types):

```
T* t1 = new T;
```

- On the GC heap (CLR types):

```
T^ t2 = gcnew T;
```

- On the stack, or as a class member:

```
T t3;
```

Pointers and Handles

```
 1 //Pointer to the native heap:
 2 NativeBox* nativeBox = new NativeBox;
 3 nativeBox->Boxify();
 4 (*nativeBox).Boxify();
 6 //Pointer to the managed (GC) heap:
 7 ManagedBox^ managedBox = gcnew ManagedBox;
 8 managedBox->Boxify();
 9 (*managedBox).Boxify();
11 //error C2440: 'initializing' :
12 //cannot convert 'cli::interior_ptr<Type>' to 'int *'
14 //int* pToTheBox = &managedBox->InTheBox;
16 //Declare a pinning pointer, and then reach
17 //for the actual address:
18 pin_ptr<int> pToTheBox = &managedBox->InTheBox;
19 int* p = pToTheBox;
```

Boxing (Value Types)

```
 1 generic <typename T>
 2 void Swap(T% first, T% second) {
 3     T temp = first;
 4     first = second;
 5     second = temp;
 6 }
 7 void BoxingAndUnboxing() {
 8     int value = 42;
 9     int^ boxed = value;
10     System::Object^ obj = boxed;
11
12     int copy = *boxed;    //Strongly-typed, no cast
13     int% refToTheBox = *boxed;
14     int newValue = 43;
15     Swap(*boxed, newValue);
16 }
```

Heap, Stack and What's in Between

Demo

Marshaling (Interop)

- Primitive types are “naturally” marshaled
- Strings are the main problem

```
1  System::String^ s = "Hello World!";
2   pin_ptr<const wchar_t> p =
3      PtrToStringChars(s);
4
5   wchar_t* unicode = p;
6   char* ansi = (char*)(void*)
7      Marshal::StringToHGlobalAnsi(s);
8
9   System::String^ s2 = gcnew System::String(ansi);
10 System::String^ s3 = gcnew System::String(unicode);
```

Marshaling Framework

- `marshal_context`
- `TTo marshal_as(TFrom)`

CLR Types in the Native World

```
 1 #include <msclr\marshal.h>
 2 #include <msclr\marshal_cppstd.h>
 3 using namespace msclr::interop;
 4
 5 class XmlInitializable {
 6 private:
 7     gcroot<XmlDocument^> _document;
 8 public:
 9     void Load(const std::string& fileName) {
10         marshal_context context;
11         XmlTextReader^ reader = gcnew XmlTextReader(
12             context.marshal_as<String^>(fileName));
13         _document = gcnew XmlDocument();
14         _document->Load(reader);
15     }
16 };
```

Native Types In The CLR

```
 1 Permutations(IEnumerable<String^>^ strings) {
 2     _strings = new vector<string>;
 3     for each (String^ s in strings)
 4         _strings->push_back(
 5                 _context.marshal_as<string>(s)); }
 6 array<String^>^ Next() {
 7     _hasNextPermutation = next_permutation(
 8          _strings->begin(), _strings->end());
 9     array<String^>^ list =
10          gcnew array<String^>(_strings->size());
11     for (it = _strings->begin();
12          it != _strings->end(); ++it) {
13         list[it - _strings->begin()] =
14                 _context.marshal_as<String^>(*it);
15     }
16     return list; }
```

Marshaling

Demo

Uniform Destruction/Finalization

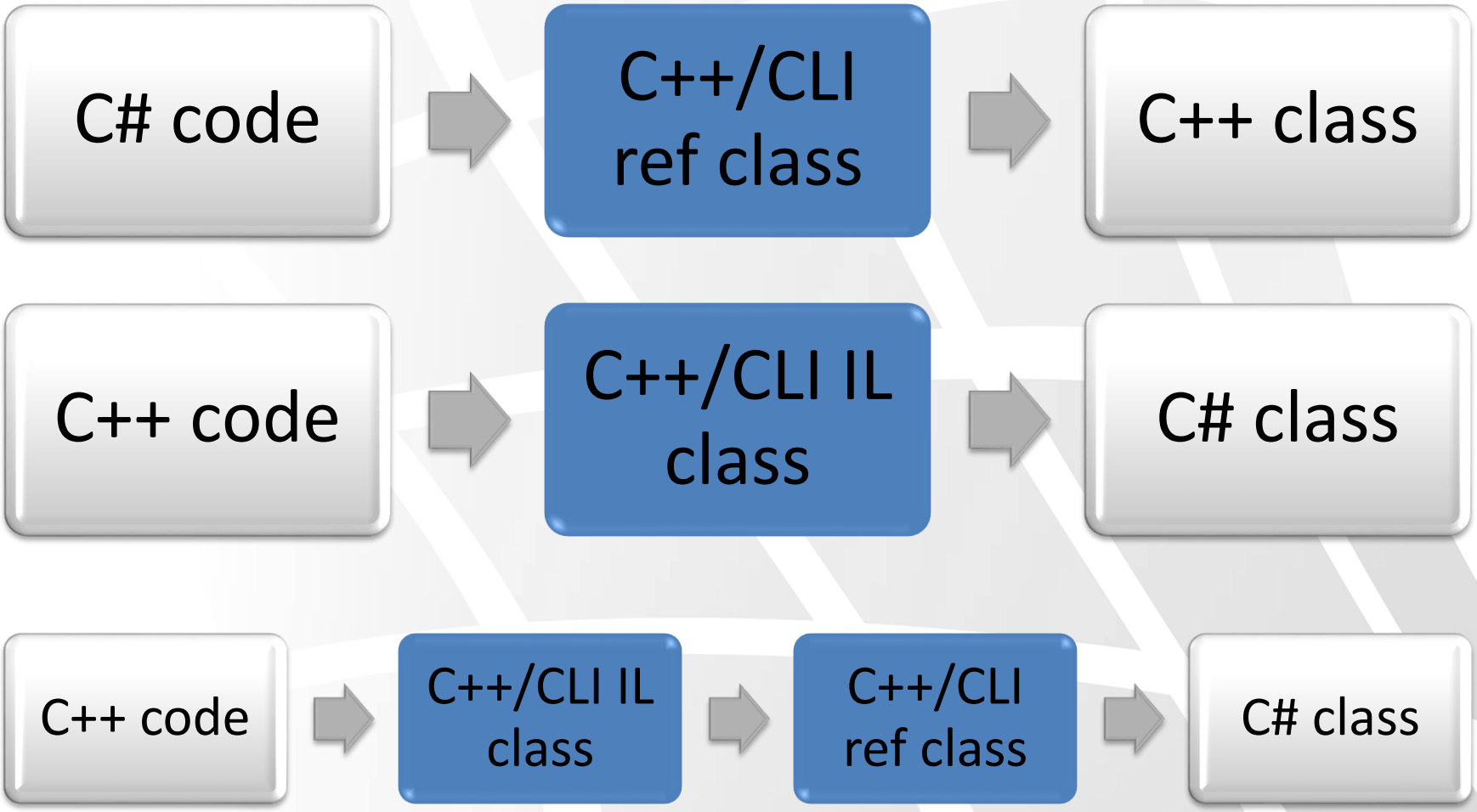
- Every type can have a destructor, ~T()
- Every type can have a finalizer, !T()

```
1  Permutations^ p1 = gcnew Permutations(EmptyArray);
2  delete p1;    //Calls the "destructor“
3
4  Permutations^ p2 = gcnew Permutations(EmptyArray);
5  //No explicit delete, so finalizer will be called later
6
7  {
8     Permutations p3(EmptyArray);
9     p3.HasNextPermutation;    //Direct access, no ->
10 }    //"destructor" called at this line
```

Destruction and Finalization

Demo

Practical Interop Scenarios



Native FileSystemWatcher
Low-Fragmentation Heap Wrapper

Lab

C++/CLI Summary

- The most powerful .NET language

- Interop is easiest: It Just Works
- Absolute control over marshaling

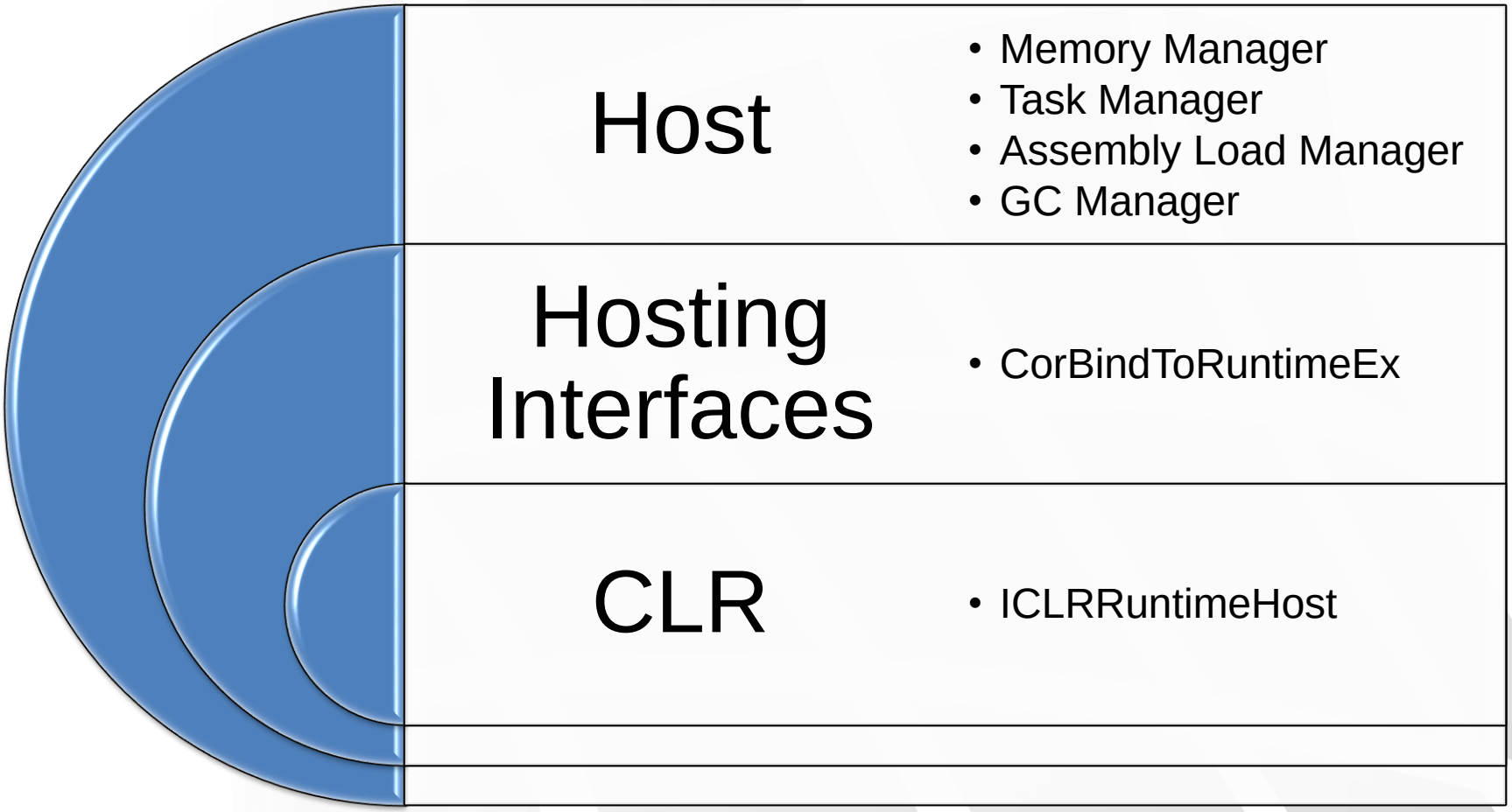
Interoperability Considerations

- P/Invoke: C-style exported DLL functions
  - Partial marshaling customization
  - Good performance
- COM Interop: COM objects
  - Hardly any marshaling customization
  - Mediocre performance
- C++/CLI: Anything C++
  - Absolute control over marshaling
  - Best performance if you know what you're doing

CLR Hosting

- Extremely powerful customization technique
- Host the CLR and tell it what to do
- The CLR relies on your services

CLR Hosting From 10,000 ft



Summary

- Platform Invoke
- COM Interop
- C++/CLI
- Overview of CLR Hosting

Questions