Foreword by Stanley B. Lippman

It is with great satisfaction that I introduce you to Stephen's excellent new book, *Pro Visual C++/CLI and the .NET 2.0 Platform*, the first detailed treatment of what has been standardized under ECMA as C++/CLI. Of course, any text, no matter how excellent, is itself incomplete, like a three-walled room. The fourth wall, in this case, is you, the reader. You complete the text by exercising the code samples, poking around with them, and finally writing your own code. That's really the only way to develop a deep understanding of this stuff. But having an experienced guide to step you through the hazards of any new language is priceless, and this is what Stephen's text accomplishes. I cannot recommend it too highly.

With Stephen's indulgence, I would like to give you a short overview of the ideas behind the language's original design and place it in the context of the design and evolution of C++ itself. The first question people ask is, "So what is C++/CLI?"

C++/CLI is a self-contained, component-based dynamic programming language that, like C# or Java, is derived from C++. Unlike those languages, however, we have worked hard to integrate C++/CLI into ISO-C++, using the historical model of evolving the C/C++ programming language to support modern programming paradigms. Historically, one can say that C++/CLI is to C++ as C++ is to C. More generally, one can view the evolution leading to C++/CLI in the following historical context:

- BCPL (Basic Computer Programming Language)
- B (Ken Thompson, original Unix work ...)
- C (Dennis Ritchie, adding type and control structure to B ...)
- C with Classes (~1979)
  - C84 (~1984) ...
  - Cfront, release E (~1984, to universities) ...
  - Cfront, release 1.0 (1985, to the world)—20th birthday !!!
- Multiple/Virtual Inheritance Programming (~1988) (MI)
- Generic Programming (~1991) (Templates)
  - ANSI C++/ISO-C++ (~1996)
- Dynamic Component Programming (~2005) (C++/CLI)

C++/CLI represents a tuple. The first term, *C++*, refers of course to the *C++ programming language* invented by Bjarne Stroustrup at Bell Laboratories. It supports a static object model that is optimized for the speed and size of its executables. It does not support runtime modification of the program other than, of course, heap allocation. It allows unlimited access to the underlying machine, but very little access to the types active in the running program, and no real access to the associated infrastructure of that program.

The third term, *CLI*, refers to the *Common Language Infrastructure*, a multitiered architecture supporting a *dynamic component* programming model. In many ways, this represents a complete reversal of the C++ object model. A runtime software layer, the virtual execution system, runs between the program and the underlying operating system. Access to the underlying machine is fairly

constrained. Access to the types active in the executing program and the associated program infrastructure—both as discovery and construction—is supported.

The second term, slash (/), represents a *binding* between C++ and the CLI.

So, a first approximation of an answer as to "What is C++/CLI?" is to say that it is a binding of the static *C++* object model with the dynamic component object model of the CLI. In short, it is how we do .NET programming using C++ rather than, say, C# or Visual Basic. Like C# and the CLI itself, C++/CLI is undergoing standardization under ECMA (and eventually under ISO).

The *common language runtime* (CLR) is the implementation of the CLI that is platform specific to the Windows operating system. Similarly, *Visual C++ 2005* is our implementation of C++/CLI.

So, as a second approximation of an answer, I would say that C++/CLI integrates the .NET programming model within C++ in the same way as, back at Bell Laboratories, we integrated generic programming using templates within the then existing C++. In both cases, both your investment in an existing C++ code base and in your existing C++ expertise are preserved. This was an essential baseline requirement of the design of C++/CLI.

What Does Learning C++/CLI Involve?

There are three aspects in the design of a CLI language that hold across all languages: (1) a mapping of language-level syntax to the underlying Common Type System (CTS); (2) the choice of a level of detail to expose the underlying CLI infrastructure to the direct manipulation of the programmer; and, (3) the choice of additional functionality to provide over that supported directly by the CLI. A fourth element of designing a CLI extension to an existing language, such as C++ or Ada, requires a fourth aspect: (4) that of integrating the managed and native type systems. We'll briefly look at an example of each in turn.

How Does C++/CLI Map to the CTS?

One aspect of programming C++/CLI is learning the underlying Common Type System, which includes three general class types:

1. A polymorphic *reference* type that is used for all class inheritance
2. A nonpolymorphic *value* type that is used for implementing concrete types requiring runtime efficiency such as the numeric types
3. An abstract *interface* type that is used for defining a set of operations common to a set of either reference or value types that implement the interface

This design aspect, the mapping of the CTS to a set of built-in language types, is common across all CLI languages, although of course the syntax varies in each CLI language. So, for example, in C#, one writes

```
abstract class Shape { ... } // C#
```

to define an abstract `Shape` base class from which specific geometric objects are to be derived, while in C++/CLI one writes

```
ref class Shape abstract { ... }; // C++/CLI
```

to indicate the exact same underlying CLI reference type. The two declarations are represented exactly the same in the underlying CIL. Similarly, in C#, one writes

```
struct Point2D { ... } // C#
```

to define a concrete `Point2D` class, while in C++/CLI one writes

```
value class Point2D { ... }; // C++/CLI
```

The family of class types supported with C++/CLI represents an integration of the CTS with the native facilities, of course, and that determined our choice of syntax. For example:

```
class native {};
value class V {};
ref class R {};
interface class I {};
```

The CTS also supports an enumeration class type that behaves somewhat differently from the native enumeration, and we provide support for both of those as well:

```
enum native { fail, pass };
enum class CLIEnum : char { fail, pass};
```

Similarly, the CTS supports its own array type that again behaves differently from the native array. And again we provide support for both:

```
int native[] = { 1,1,2,3,5,8 };
array<int>^ managed = { 1,1,2,3,5,8 };
```

It is not true to think of any one CLI language as closer to or more nearly a mapping to the underlying CTS than is another. Rather, each CLI language represents a view into the underlying CTS object model.

What Level of Detail of the CLI Does C++/CLI Expose?

The second design aspect reflects the level of detail of the underlying CLI implementation model to incorporate into the language. How does one go about determining this? Essentially, we need to ask these questions:

- What are the kinds of problems the language is likely to be tasked to solve? We must make sure the language has the tools necessary to do this.
- What are the kinds of programmers the language is likely to attract?

Let's look at an example: the issue of value types occurring on the managed heap. Value types can find themselves on the managed heap in a number of circumstances:

- Implicit boxing
  - We assign an object of a value type to an Object.
  - We invoke a virtual method through a value type that is not overridden.
- When a value type serves as a member of a reference class type
- When a value type is being stored as the element type of a CLI array

The design question a CLI language has to ask is, "Should we allow the programmer to manipulate the address of a value type of this sort?"

What are the issues?

Any object located on the managed heap is subject to relocation during the compaction phase of a sweep of the garbage collector. Any pointers to that object must be tracked and updated by the runtime; the programmer has no way to manually track it herself. Therefore, if we were to allow the programmer to take the address of a value type potentially resident on the managed heap, we would need to introduce a tracking form of pointer in addition to the existing native pointer.

What are the trade-offs to consider? On the one hand, simplicity and safety.

- Directly introducing support in the language for one or a family of tracking pointers makes it a more complicated language. By not supporting this, we expand the available pool of programmers by requiring less sophistication.
- Allowing the programmer access to these ephemeral value types increases the possibility of programmer error—she may purposely or by accident do bad things to the memory. By not supporting this, we create a potentially safer runtime environment.

On the other hand, efficiency and flexibility.

- Each time we assign the same Object with a value type, a new boxing of the value occurs. Allowing access to the boxed value type allows in-memory update, which may provide significant performance ...
- Without a form of tracking pointer, we cannot iterate over a CLI array using pointer arithmetic. This means that the CLI array cannot participate in the STL iterator pattern and work with the generic algorithms. Allowing access to the boxed value type allows significant design flexibility.

We chose in C++/CLI to provide a collection of addressing modes that handle value types on the managed heap.

```
int ival = 1024;

// int^ provides a tracking handle for
//      direct read/write access to a boxed value type ...
int^ boxedi = ival;

array<int>^ ia = gcnew array<int>{1,1,2,3,5,8};

// interior_ptr<T> supports indexing into the GC heap ...
interior_ptr<int> begin = &ia[0];

value struct smallInt { int m_ival; ... } si;
pin_ptr<int> ppi = &si.m_ival;
```

We imagine the C++/CLI programmer to be a sophisticated system programmer tasked with providing infrastructure and organizationally critical applications that serve as the foundation over which a business builds its future. She must address both scalability and performance concerns and must therefore have a system-level view into the underlying CLI. The level of detail of a CLI language reflects the face of its programmer.

Complexity is not in itself a negative quality. Human beings, for example, are more complicated than single-cell bacteria, but that is, I think we all agree, not a bad thing. When the expression of a simple concept is complicated, that is a bad thing. In C++/CLI, we have tried to provide an elegant expression to a complex subject matter.

What Does C++/CLI Add Over That of the CLI?

A third design aspect is a language-specific layer of functionality over that directly supported by the CLI. This may require a mapping between the language-level support and the underlying implementation model of the CLI. In some cases, this just isn't possible because the language cannot intercede with the behavior of the CLI. One example of this is the virtual function resolution in the constructor and destructor of a base class. To reflect ISO-C++ semantics in this case would require a resetting of the virtual table within each base class constructor and destructor. This is not possible because virtual table handling is managed by the runtime and not the individual language.

So this design aspect is a compromise between what we might wish to do, and what we find ourselves able to do. The three primary areas of additional functionality provided by C++/CLI are the following:

- A form of *Resource Acquisition is Initialization* (RAII) for reference types. In particular, to provide an automated facility for what is referred to as *deterministic finalization* of garbage collected types that hold scarce resources.
- A form of deep-copy semantics associated with the C++ copy constructor and copy assignment operator; however, this could not be extended to value types.
- Direct support of C++ templates for CTS types in addition to the CLI generic mechanism—this had been the topic of my original first column. In addition, we provide a verifiable version of the Standard Template Library for CLI types.

Let's look at a brief example: the issue of deterministic finalization.

Before the memory associated with an object is reclaimed by the garbage collector, an associated `Finalize()` method, if present, is invoked. You can think of this method as a kind of super-destructor since it is not tied to the program lifetime of the object. We refer to this as *finalization*. The timing of just when or even whether a `Finalize()` method is invoked is undefined. This is what is meant when we say that garbage collection exhibits *nondeterministic finalization*.

Nondeterministic finalization works well with dynamic memory management. When available memory gets sufficiently scarce, the garbage collector kicks in and things pretty much just work. Nondeterministic finalization does not work well, however, when an object maintains a critical resource such as a database connection, a lock of some sort, or perhaps native heap memory. In this case, we would like to release the resource as soon as it is no longer needed. The solution currently supported by the CLI is for a class to free the resources in its implementation of the `Dispose()` method of the `IDisposable` interface. The problem here is that `Dispose()` requires an explicit invocation, and therefore is liable not to be invoked.

A fundamental design pattern in C++ is spoken of as Resource Acquisition is Initialization. That is, a class acquires resources within its constructor. Conversely, a class frees its resources within its destructor. This is managed automatically within the lifetime of the class object.

This is what we would like to do with reference types in terms of the freeing of scarce resources:

- Use the destructor to encapsulate the necessary code for the freeing of any resources associated with the class.
- Have the destructor automatic invocation tied with the lifetime of the class object.

The CLI has no notion of the class destructor for a reference type. So the destructor has to be mapped into something else in the underlying implementation. Internally, then, the compiler does the following transformations:

- The class has its base class list extended to inherit from the `IDisposable` interface.
- The destructor is transformed into the `Dispose()` method of `IDisposable`.

That gets us half the way to our goal. We still need a way to automate the invocation of the destructor. A special stack-based notation for a reference type is supported; that is, one in which its lifetime is associated within the scope of its declaration. Internally, the compiler transforms the notation to allocate the reference object on the managed heap. With the termination of the scope, the compiler inserts an invocation of the `Dispose()` method—the user-defined destructor. Reclamation of the actual memory associated with the object remains under the control of the garbage collector.

Let's look at a code example.

```
ref class Wrapper {
    Native *pn;
public:
    // resource acquisition is initialization
    Wrapper( int val ) { pn = new Native( val ); }

    // this will do our disposition of the native memory
    ~Wrapper(){ delete pn; }

    void mfunc();
protected:

    // an explicit Finalize() method - as a failsafe ...
    ! Wrapper() { delete pn; }
};

void f1()
{
   // normal treatment of a reference type ...
   Wrapper^ w1 = gcnew Wrapper( 1024 );

   // mapping a reference type to a lifetime ...
   Wrapper w2( 2048 ); // no ^ token !

   // just illustrating a semantic difference ...
   w1->mfunc(); w2.mfunc();

  // w2 is disposed of here
}

//
// ... later, w1 is finalized at some point, maybe ...
```

C++/CLI is not just an extension of C++ into the managed world. Rather, it represents a fully integrated programming paradigm similar in extent to the earlier integration of the multiple inheritance and generic programming paradigms into the language. I think the team has done an outstanding job.

Integrating C++/CLI with ISO-C++

The type of a string literal, such as "Pooh", is treated differently within C++/CLI; it is more nearly a kind of System::String than a *C-style* character string pointer. This has a visible impact with regard to the resolution of overload functions. For example:

```
public ref class R {
public:
  void foo( System::String^ ); // (1)
  void foo( std::string );     // (2)
  void foo( const char* );     // (3)
};

void bar( R^ r )
{
  // which one?
  r->foo( "Pooh" );
}
```

In ISO-C++, this resolves to instance (3)—a string literal is more nearly a kind of constant pointer to character than it is an ISO-C++ standard library string type. Under C++/CLI, however, this call resolves to (1)—a string literal is now more nearly a kind of System::String than pointer to character. The type of a string literal is treated differently within C++/CLI. It has been designed to be more nearly a kind of System::String than a C-style character string pointer.

```
void foo( System::String^ ); // (1)
void foo( std::string );           // (2)
void foo( const char* );         // (3)

void bar( R^ r ){  r->foo( "Pooh" ); }  // which foo?

ISO-C++: // (3) is invoked ...
C++/CLI: // (1) is invoked ...
```

So, What Did You Say About C++/CLI?

C++/CLI represents an integration of native and managed programming. In this iteration, we have done that through a kind of separate but equal community of source-level and binary elements:

- Mixed mode: source-level mix of native and CTS types plus binary mix of native and CIL object files. (Compiler switch: \clr.)
- Pure mode: source-level mix of native and CTS types. All compiled to CIL object files. (Compiler switch: \clr:pure.)
- Native class can hold CTS types through a special wrapper class only.
- CTS classes can hold native types only as pointers.

Of course, the C++/CLI programmer can also choose to program with the .NET managed types only, and in this way provide verifiable code, using the \clr:safe Visual C++ compiler switch.