Foundations of C++/CLI

The Visual C++ Language for .NET 3.5

Gordon Hogenson

Apress®

Foundations of C++/CLI: The Visual C++ Language for .NET 3.5

ISBN-13 (pbk): 978-1-4302-1023-8

ISBN-13 (electronic): 978-1-4302-1024-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matthew Moodie
Technical Reviewer: Damien Watkins
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Kylie Johnston
Copy Editor: Ami Knox
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositors: Susan Glinert, Pat Christenson
Proofreader: April Eddy
Indexer: John Collin
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales—eBook Licensing web page at http://www.apress.com/info/bulksales.

The source code for this book is available to readers at http://www.apress.com.

To my brother Kirk, a man of exemplary character and delightful humor.

Contents at a Glance

Contents

Foreword by Brandon Bray

Few things excite me more than thinking about the potential of software. It sounds contrived, but it's true. To that end, programmer productivity is essential to building great software. It's easy to look at C++ and begin lauding it for powerful techniques like templates and low-level control, and likewise C++ is vilified for complex preprocessing and loose type safety, among other things. Yet C++ is one of the most productive programming languages on the market.

Early on in the design of C++, a conscious decision was made to make C++ highly compatible with C. That philosophy persisted as the language evolved—*existing code matters!* When new system technologies like COM appeared, C++ directly supported them without forcing programmers to discard code already written. So, when .NET came about, there was no question that C++ should enable programmers to leverage value from their existing code while using the .NET Framework.

Of course, the first few attempts at integrating concepts from .NET into C++ proved to be a challenging task. Nevertheless, the first release of "Managed C++" laid the groundwork for what was to come. Customers taught us that syntax matters; a good language is a balance between utility and elegance.

At this point, I started involvement in the project. As the person tasked with writing the specification, it is easy to associate me with the development of C++/CLI. Yet this truly was a collaborative effort that involved intense discussion, experimentation, feedback, iteration, and advocacy from hundreds of people. Developers, testers, customers, book authors, standards advocates, bloggers, and even journalists gave feedback that changed C++/CLI in some way.

The end result is a language that enables developers to maximize their usage of C++ code in existence while giving them the freedom to explore the possibilities of the .NET Framework. C++/CLI is a language that strengthens the value of both managed and native code. Complexity is still a concern, but I am satisfied that the essential concepts of managed code show through in an elegant manner.

I am excited that you have Gordon's book in your hands. With it, you will take away incredibly valuable skills. After all, the software you create has the potential to change the world.

Brandon Bray
Senior Program Manager, Microsoft

Foreword to the First Edition by Stanley B. Lippman

A person standing on the side of a river shouts to someone on the opposite bank: "How do you get to the other side?" The second person replies: "You are on the other side."

—Chris Gosden

C++/CLI is a binding of C++ to Microsoft's .NET programming environment. It integrates ISO C++ with the Unified Type System (UTS) of the Common Language Infrastructure (CLI). It supports both source-level and binary interoperability between native and managed C++. As the Gosden quote suggests, it is how one gets to the other side, regardless of where you happen to be standing. The actual details of how you do this are covered in Gordon's fine text.

In primitive societies and adolescent fantasy novels, such as *The Lord of the Rings* (which, along with *Remembrance of Things Past*, is one of *my* favorite books), names have a kind of magical aura to them—they need to be handled with extreme care and protected. The same holds true in computer science, apparently—or at least within Microsoft. Although you hold in your hand the first book devoted solely to C++/CLI, I couldn't for the life of me find any specific reference to C++/CLI in the Visual Studio 2005 release—at least not in the Visual C++ IDE, in order to open a C++/CLI project, or in the "What's New" section of the documentation. This whole notion of binding C++ to .NET has a sort of fantasy aspect to it that has clung to it since the original Managed Extensions to C++ in the Visual Studio .NET release of 2001. C++/CLI is the noncompatible and more elegant replacement for the Managed Extensions. It is how we program .NET using what the book's subtitle calls "the Visual C++ Language for .NET." That's what Gordon's book will teach you how to do.

As Gordon states in his introduction, C++/CLI represents an evolution of C++. This does not, of course, imply that C++/CLI is a better language than C++; rather, C++/CLI is better adapted to the current and future computing environment that we work in. If you are a Visual C++ programmer with legacy "native applications" and need to move or extend these applications to .NET, C++/CLI is an essential tool for your survival, and Gordon's text is an essential first step to mastering this tool.

An aspect of evolution is an increase in structural complexity, and this, too, is reflected in C++/CLI: knowing C++ may or may not be a help in understanding C++/CLI! For example, there is no such thing as a destructor in .NET, so although the syntax resembles that of the native C++ destructor, its behavior is oddly counterintuitive: you simply can't fully understand its operation by its analogous form. And this is where Gordon's text becomes invaluable both as a tutorial and a desktop reference. It is for this reason that I highly recommend it.

Stanley B. Lippman
Former Architect, Visual C++

Foreword by Herb Sutter

A Design Rationale for C++/CLI

—Excerpted from "A Design Rationale for C++/CLI" by Herb Sutter. (Full text available online at `http://www.gotw.ca/publications/C++CLIRationale.pdf`*.)*

1 Overview

A multiplicity of libraries, runtime environments, and development environments are essential to support the range of C++ applications. This view guided the design of C++ as early as 1987; in fact, it is older yet. Its roots are in the view of C++ as a general-purpose language.

—B. Stroustrup (*Design and Evolution of C++*, Addison-Wesley Professional, 1994, p. 168)

C++/CLI was created to enable C++ use on a major runtime environment, ISO CLI (the standardized subset of .NET).

A technology like C++/CLI is essential to C++'s continued success on Windows in particular. CLI libraries are the basis for many of the new technologies on the Windows platform, including the WinFX class library shipping with Windows Vista, which offers over 10,000 CLI classes for everything from web service programming (Communication Foundation, WCF) to the new 3D graphics subsystem (Presentation Foundation, WPF). Languages that do not support CLI programming have no direct access to such libraries, and programmers who want to use those features are forced to use one of the 20 or so other languages that do support CLI development. Languages that support CLI include COBOL, C#, Eiffel, Java, Mercury, Perl, Python, and others; at least two of these have standardized language-level bindings.

C++/CLI's mission is to provide direct access for C++ programmers to use existing CLI libraries and create new ones, with little or no performance overhead, with the minimum amount of extra notation, and with full ISO C++ compatibility.

1.1 Key Goals

- *Enable C++ to be a first-class language for CLI programming.*
  - Support important CLI features, at minimum those required for a CLS consumer and CLS extender: CLI defines a Common Language Specification (CLS) that specifies the subsets of CLI that a language is expected to support to be minimally functional for consuming and/or authoring CLI libraries.
  - Enable C++ to be a systems programming language on CLI: a key existing strength of C++ is as a systems programming language, so extend this to CLI by leaving no room for a CLI language lower than C++(besides ILASM).
- Use the fewest possible extensions.
  - Require zero use of extensions to compile ISO C++ code to run on CLI: C++/CLI requires compilers to make ISO C++ code "just work"—no source code changes or extensions are needed to compile C++ code to execute on CLI, or to make calls between code compiled "normally" and code compiled to CLI instructions.
  - Require few or no extensions to consume existing CLI types: to use existing CLI types, a C++ programmer can ignore nearly all C++/CLI features and typically writes a sprinkling of `gcnew` and ^. Most C++/CLI extensions are used only when authoring new CLI types.
  - Use pure conforming extensions that do not change the meaning of existing ISO C++ programs and do not conflict with ISO C++ or with C++0x evolution: this was achieved nearly perfectly, including for macros.
- Be as orthogonal as possible.
  - Observe the principle of least surprise: if feature X works on C++ types, it should also seamlessly work on CLI types, and vice versa. This was mostly achieved, notably in the case of templates, destructors, and other C++ features that do work seamlessly on CLI types; for example, a CLI type can be templated and/or be used to instantiate a template, and a CLI generic can match a template parameter.

Some unifications were left for the future; for example, a contemplated extension that the C++/CLI design deliberately leaves room for is to use `new` and * to (semantically) allocate CLI types on the C++ heap, making them directly usable with existing C++ template libraries, and to use `gcnew` and ^ to (semantically) allocate C++ types on the CLI heap. Note that this would be highly problematic if C++/CLI had not used a separate `gcnew` operator and ^ declarator to keep CLI features out of the way of ISO C++.

1.2 Basic Design Forces

Four main programming model design forces are mentioned repeatedly in this paper:

1. It is necessary to add language support for a key feature that semantically cannot be expressed using the rest of the language and/or must be known to the compiler.

> *Classes can represent almost all the concepts we need. . . . Only if the library route is genuinely infeasible should the language extension route be followed.*
>
> —B. Stroustrup (*Design and Evolution of C++*, p. 181)

In particular, a feature that unavoidably requires special code generation must be known to the compiler, and nearly all CLI features require special code generation. Many CLI features also require semantics that cannot be expressed in C++. Libraries are unquestionably preferable wherever possible, but either of these requirements rules out a library solution. Note that language support remains necessary even if the language designer smoothly tries to slide in a language feature dressed in library's clothing (i.e., by choosing a deceptively library-like syntax). For example, instead of

```
property int x; // A: C++/CLI syntax
```

the C++/CLI design could instead have used (among many other alternatives) a syntax like

```
property<int> x; // B: an alternative library-like syntax
```

and some people might have been mollified, either because they looked no further and thought that it really was a library, or because they knew it wasn't a library but were satisfied that it at least looked like one. But this difference is entirely superficial, and nothing has really changed—it's still a language feature and a language extension to C++, only now a deceitful one masquerading as a library (which is somewhere between a fib and a bald-faced lie, depending on your general sympathy for magical libraries and/or grammar extensions that look like libraries).

In general, even if a feature is given library-like syntax, it is still not a true library feature when

- The name is recognized by the compiler and given special meaning (e.g., it's in the language grammar, or it's a specially recognized type) and/or
- The implementation is "magical."

Either of these make it something no user-defined library type could be. Note that, in the case of surfacing CLI properties in the language, at least one of these must be true even if properties had been exposed using syntax like B.

Therefore, choosing a syntax like B would not change anything about the technical fact of language extension, but only the political perception. This approach amounts to dressing up a language feature with library-like syntax that pretends it's something that it can't be. C++'s tradition is to avoid magic libraries and has the goal that the C++ standard library should be implementable in C++ without compiler collusion, although it allows for some functions to be intrinsics known to the compiler or processor. C++/CLI prefers to follow C++'s tradition, and it uses magical types or functions only in four isolated cases: `cli::array`, `cli::interior_ptr`, `cli::pin_ptr`, and `cli::safe_cast`. These four can be viewed as intrinsics—their implementations are provided by the CLI runtime environment and the names are recognized by the compiler as tags for those CLI runtime facilities.

2. It is important not only to hide unnecessary differences, but also to expose essential differences.

I try to make significant operations highly visible.

—B. Stroustrup (*Design and Evolution of C++*, p. 119)

First, an unnecessary distinction is one where the language adds a feature or different syntax to make something look or be spelled differently, when the difference is not material and could have been "papered over" in the language while still preserving correct semantics and performance. For example, CLI reference types can never be physically allocated on the stack, but C++ stack semantics are very powerful, and there is no reason not to allow the lifetime semantics of allocating an instance of a reference type `R` on the stack and leveraging C++'s automatic destructor call semantics. C++/CLI can, and therefore should, safely paper over this difference and allow stack-based semantics for reference type objects, thus avoiding exposing an unnecessary distinction. Consider this code for a reference type `R`:

```
void f()
{
   R r;// OK, conceptually allocates the R on the stack
   r.SomeFunc(); // OK, use value semantics
   ...
} // destroy r here
```

In the programming model, `r` is on the stack and has normal C++ stack-based semantics. Physically, the compiler emits something like the following:

```
// f, as generated by the compiler
void f()
{
   R^ r = gcnew R; // actually allocated on the CLI heap
   r->SomeFunc();// actually uses indirection
   ...
   delete r;// destroy r here (memory is reclaimed later)
}
```

Second, it is equally important to avoid obscuring essential differences, specifically not try to "paper over" a difference that actually matters but where the language fails to add a feature or distinct syntax.

For example, although CLI object references are similar to pointers (e.g., they are an indirection to an object), they are nevertheless semantically not the same because they do not support all the operations that pointers support (e.g., they do not support pointer arithmetic, stable values, or reliable comparison). Pretending that they are the same abstraction, when they are not and cannot be, causes much grief. One of the main flaws in the Managed Extensions design is that it tried to reduce the number of extensions to C++ by reusing the * declarator, where `T*` would implicitly mean different things depending the type of T—but three different and semantically incompatible things, lurking together under a single syntax.

The road to unsound language design is paved with good intentions, among them the papering over of essential differences.

3. Some extensions actively help avoid getting in the way of ISO C++ and C++0x evolution.

Any compatibility requirements imply some ugliness.

—B. Stroustrup (*Design and Evolution of C++*, p. 198)

A real and important benefit of extensions is that using an extension that the ISO C++ standards committee (WG21) has stated it does not like and is not interested in can be the best way to stay out of the way of C++0x evolution, and in several cases this was done explicitly at WG21's direction.

For example, consider the extended `for` loop syntax: C++/CLI stayed with the syntax `for each( T t in c )` after consulting the WG21 evolution working group at the Sydney meeting in March 2004 and other meetings, where EWG gave the feedback that they were interested in such a feature but they disliked both the `for each` and `in` syntax and were highly likely never to use it, and so directed C++/CLI to use the undesirable syntax in order to stay out of C++0x's way. (The liaisons noted that if in the future WG21 ever adopts a similar feature, then C++/CLI would want to drop its syntax in favor of the WG21-adopted syntax; in general, C++/CLI aims to track C++0x.)

Using an extension that WG21 might be interested in, or not using an extension at all but adding to the semantics of an existing C++ construct, is liable to interfere with C++0x evolution by accidentally constraining it. For another example, consider C++/CLI's decision to add the `gcnew` operator and the ^ declarator. . . . Consider just the compatibility issue: by adding an operator and a declarator that are highly likely never to be used by ISO C++, C++/CLI avoids conflict with future C++ evolution (besides making it clear that these operations have nothing to do with the normal C++ heap). If C++/CLI had instead specified a `new (gc)`or `new (cli)` "placement new" as its syntax for allocation on the CLI heap, that choice could have conflicted with C++0x evolution that might want to provide additional forms of placement new. And, of course, using a placement syntax could and would also conflict with existing code that might already use these forms of placement new—in particular, `new (gc)` is already used with the popular Boehm collector.

4. Users rely heavily on keywords, but that doesn't mean the keywords have to be reserved words.

My experience is that people are addicted to keywords for introducing concepts to the point where a concept that doesn't have its own keyword is surprisingly hard to teach. This effect is more important and deep-rooted than people's vocally expressed dislike for new keywords. Given a choice and time to consider, people invariably choose the new keyword over a clever workaround.

—B. Stroustrup (*Design and Evolution of C++*, p. 119)

When a language feature is necessary, programmers strongly prefer keywords. Normally, all C++ keywords are also reserved words, and taking a new one would break code that is already using that word as an identifier (e.g., as a type or variable name).

C++/CLI avoids adding reserved words so as to preserve the goal of having pure extensions, but it also recognizes that programmers expect keywords. C++/CLI balances these requirements by adding keywords where most are not reserved words and so do not conflict with user identifiers.

For a related discussion, see also my blog article "C++/CLI Keywords: Under the hood" (November 23, 2003).

- *Spaced keywords*: These are reserved words, but cannot conflict with any identifiers or macros that a user may write because they include embedded whitespace (e.g., `ref class`).
- *Contextual keywords*: These are special identifiers instead of reserved words. Three techniques were used:
  1. Some do not conflict with identifiers at all because they are placed at a position in the grammar where no identifier can appear (e.g., `sealed`).
  2. Others can appear in the same grammar position as a user identifier, but conflict is avoided by using a different grammar production or a semantic disambiguation rule that favors the ISO C++ meaning (e.g., `property`, `generic`), which can be informally described as the rule "If it can be a normal identifier, it is."
  3. Four "library-like" identifiers are considered keywords when name lookup finds the special marker types in namespace `cli` (e.g., `pin_ptr`).

Note these make life harder for compiler writers, but that was strongly preferred in order to achieve the dual goals of retaining near-perfect ISO C++ compatibility by sticking to pure extensions and also being responsive to the widespread programmer complaints about underscores.

1.3 Previous Effort: Managed Extensions

C++/CLI is the second publicly available design to support CLI programming in C++. The first attempt was Microsoft's proprietary Managed Extensions to C++ (informally known as "Managed C++"), which was shipped in two releases of Visual C++ (2002 and 2003) and continues to be supported in deprecated mode in Visual C++ 2005.

Because the Managed Extensions design deliberately placed a high priority on C++ compatibility, it did two things that were well-intentioned but that programmers objected to:

- The Managed Extensions wanted to introduce as few language extensions as possible, and ended up reusing too much existing but inappropriate C++ notation (e.g., * for pointers). This caused serious problems where it obscured essential differences, and the design for overloaded syntaxes like * was both technically unsound and confusing to use.
- The Managed Extensions scrupulously used names that the C++ standard reserves for C++ implementations, notably keywords that begin with a double underscore (e.g., __gc). This caused unexpectedly strong complaints from programmers, who made it clear that they hated writing double underscores for language features.

Many C++ programmers tried hard to use these features, and most failed. Having the Managed Extensions turned out to be not significantly better for C++ than having no CLI support at all. However, the Managed Extensions did generate much direct real-world user experience with a shipping product about what kinds of CLI support did and didn't work, and why; and this experience directly informed C++/CLI.

Herb Sutter
Architect

About the Author

GORDON HOGENSON grew up in Fairbanks, Alaska, and retains the independent spirit and love of nature he learned there. Long torn between a love of writing and a love of science, he wrote a fantasy novel in high school called *Phalshazhaln* and then went on to study chemistry at Harvey Mudd College, intern in chemical physics at the University of Oregon, and work toward a Ph.D. in physical chemistry at the University of Washington, when he published a paper with William P. Reinhardt in the Journal of Chemical Physics on computational methods combining quantum mechanics and thermodynamics, as well as an article on a meditation technique for the first issue of *The Resonance Project*, later called *Trip Magazine*, a journal for the psychedelic subculture.

Supported by fellowships from Connie Ringold and the U.S. Department of Energy, he studied computational science and pursued attempts to bring together diverse ideas in the spirit of natural philosophy. He spent his free time studying the controversies at the edges of science. He began working at Microsoft in 1997 as a tester, technical writer, and manager on several Visual Studio languages including J++, C#, C++, and, more recently, F#. Gordon met his wife, Jeni, while they searched the night sky near Mt. Rainier for signs of life beyond Earth as members of CSETI, an organization devoted to furthering our understanding of extraterrestrial life. His current pastimes include raising goats on his farm near Duvall, Washington, planning a permaculture garden, and dreaming of self-sufficiency on the land.

About the Technical Reviewer

DAMIEN WATKINS is a program manager on the Visual C++ team at Microsoft. His main areas of responsibility are language conformance, the CRT and MFC libraries, and concurrency. His main area of interest is the design and implementation of component architectures. His first book, *Programming in the .NET Environment* (Addison-Wesley, 2003), coauthored with Mark Hammond and Brad Abrams, describes the architecture and goals of the .NET Framework. Prior to joining the Visual C++ Team, Damien was a member of the External Research Office at Microsoft Research Cambridge. Damien has presented tutorials, seminars, and workshops on COM/DCOM, CORBA, and the .NET Framework at numerous events, including ECOOP 2004, OOPSLA 2003, OOPSLA 2002, SIGCSE 2002, and the Microsoft Research Faculty Summit 2001.

Acknowledgments

This book would never have been possible had it not been for the constant support of Jeni, my lovely wife. I am very grateful to her for her patience with me during the project and for generally being such an inspiring presence in my life. I also want to heartily thank Damien Watkins, whose support, tough technical editing, humor, and encouragement all helped this text come together. I was also fortunate enough to have a technical review of Chapter 12 by Nikola Dudar, formerly of the Visual C++ PM team, whose detailed knowledge of STL/CLR was much appreciated. The text also benefited greatly from feedback from many Microsoft employees who devoted their time and attention to reviewing the first edition: Arjun Bijanki, whose knowledge of C++/CLI helped make the text much more accurate; Martin Chisholm, who printed and read the text very carefully while on a bike trip; John Svitak, whose attention to detail really helped improve the polish; Kirill Kobelev, who pointed out errors and omissions in the radioactivity example; Thomas Petchel, who found several programming errors and had many other good suggestions; Yves Dolce, whose familiarity with developer problems helped make the book more practical; Peter-Michael Osera, who pointed out many subtleties and asked very good questions; Ron Pihlgren, who pointed out misleading statements and questionable assertions in Chapters 3 and 13; Bob Davidson, who despite his demanding schedule managed to provide feedback on the book; Ann Beebe, who allowed me to have a flexible work schedule so I could work on the text; and Chuck Bell, who had some great ideas on the exceptions discussion in Chapter 13. I also want to thank Matt Moodie, Kylie Johnston, Ami Knox, and Kelly Winquist at Apress for their patience and help getting this into print. I want to thank Brandon Bray for his comments in the foreword for this edition, and Herb Sutter for permission to use his words that give insights into the design of the language extensions, and finally, Stan Lippman, whose idea this was and without whom none of this would ever have happened.

Introduction

Thank you for picking up this book. In it I present the C++/CLI extensions to the C++ computer programming language, a significant development in the long history of the C and C++ programming languages.

Why extend C++? C++ has evolved over many years; it is used by millions of developers worldwide. The nature of C++ has been to grow as programming paradigms evolve. After all, it was the desire to extend the C language to support object-oriented concepts that prompted Bjarne Stroustrup and his colleagues at Bell Labs to develop "C with classes." Many of the new language features that have come along have been reflected in the C++ language, such as templates, runtime type information, and so on; they have enhanced the richness (and complexity) of the language. The features added to C++ by C++/CLI are no different. C++/CLI provides a new set of extensions to the C++ language to support programming concepts such as component-based software development, garbage collection, and interoperability with other languages that run on a common virtual machine, along with other useful features.

The CLI, or Common Language Infrastructure, is a standard adopted by ECMA International. The CLI defines a virtual machine and enables rich functionality in languages that target the virtual machine, as well as a framework of libraries that provide additional support for the fundamentals of programming against the CLI virtual machine. Collectively, these libraries and the platform constitute the *infrastructure* of the CLI. It's a *common language* infrastructure because a wide variety of languages can target that infrastructure.

The name "C++/CLI" refers to a standard that describes extensions to the C++ language that allow C++ programmers to program against a CLI virtual machine.

Microsoft's implementation of the CLI standard is called the CLR, or common language runtime, or the .NET Developer Platform (NDP). The libraries Microsoft provides that implement the CLI standard are collectively known as the .NET Framework, although the .NET Framework also includes other libraries that are not part of the CLI standard. There are several other implementations of the CLI, including the .NET Compact Framework (`http://msdn.microsoft.com/netframework/programming/netcf`), the Mono Project (`http://www.mono-project.com`), and dotGNU Portable.NET (`http://dotgnu.org`). Visual C++ 2005 is the first release of Visual C++ that supports C++/CLI; Visual C++ 2008 expands that support by adding support for a version of the Standard Template Library that is compatible with C++/CLI, and better support for marshaling of data between native and managed types.

First, let's address the issue of what the term "C++/CLI" means in the technical sense. C++ is a well-known language. While some might quibble over standards conformance, C++ is essentially the language design captured by the ANSI/ISO standard in the late 1990s. Purists will say that C++/CLI is a set of language bindings to the CLI standard, not a language in and of itself. ECMA has adopted C++/CLI as a standard itself, and it is in the process of being submitted to the appropriate ISO working group. The C++/CLI language is an approximate superset of the C++ language, so if you drop all the support for the CLI from the language, you're left with C++. This means that almost any C++ program is automatically supported as a C++/CLI program, just one that doesn't refer to any of the additional functionality provided by the CLI.

Why C++/CLI?

C++/CLI was created by Microsoft to be a more friendly programming language than its predecessor, Managed Extensions for C++. Microsoft had created the CLR, and the C++ team at Microsoft had devised a syntax that provided C++ programmers with a way to target the CLR. The first release of Visual Studio to support the CLR was Visual Studio .NET 2002. The syntax that was provided with Visual Studio .NET 2002 was constrained by the desire to adhere as much as possible to the existing C++ standard, the ISO C++ Standard. According to this standard, any extensions to a language had to conform to the rules for language extensions—among other constraints, this meant keywords had to begin with a double underscore (__). Thus, Managed Extensions for C++ provided a very clumsy syntax for targeting the CLR. In order to create a "managed" pointer (one that refers to an object that is garbage collected), one used syntax as follows:

```
int __gc * ptr;
```

The managed pointers were referred to as "__gc pointers." Similarly, in order to declare a managed class, one used the __gc keyword as a modifier:

```
__gc class C { ... };
```

and to declare an interface (a concept that does not exist as a specific language feature in C++), one had to use the syntax

```
__interface I { ... };
```

There were other cases of keywords added with double underscores as well. All in all, the syntax was cumbersome.

And not just because of the double underscores, but also because Managed Extensions for C++ did not provide natural support for several key concepts of the CLR, such as properties, events, automatic boxing, and so on. All this meant that C++ programmers did not enjoy programming in Managed Extensions for C++.

Programming should be fun. Language is more than just a utilitarian concept. After all, many people spend their entire day programming. Why should they hobble along with a difficult extension when they could be using a clean, crisp language that makes programming easy and fun? The C++ team at Microsoft recognized that in order to make C++ programming enjoyable and aesthetically pleasing, as well as to take full advantage of the CLR, the syntax had to change. And that meant taking the radical step of departing from the ISO C++ Standard.

However, Microsoft had made the decision to work through standards bodies, and if it was going to depart from the ISO C++ Standard, rather than being "nonstandard," it was felt that a new standard was needed. The C++/CLI standard was born.

The new language was designed with ease of use in mind and was intended to be a breath of fresh air. It should be a great relief to anyone who has tried to use Managed Extensions for C++.

C++/CLI As a .NET Language

Since the first edition of this book was released, Microsoft's view of the position of C++/CLI in the family of .NET Languages has been clarified. A few years ago, it was not clear whether C++/CLI would be treated as an equal alternative to C# or Visual Basic (VB) .NET for creating .NET applications. Microsoft realized that it would be a huge investment to implement all of the

productivity features available in C# and VB .NET, such as visual designers and the like, equally for C++/CLI. The issue came to a head early in the Visual Studio 2008 product cycle with the upcoming release of the XAML designers for WPF. By this time, it was clear that the industry was looking to C# as the language of choice for the development of new .NET applications, and that C++/CLI was being used mostly when mixing native and managed code in the same application. As a result, there is no designer support in Visual Studio 2008 for creating WPF applications in C++/CLI. More recently, the role of C++/CLI has been clarified. C++/CLI is primarily intended for interoperability ("interop") between native and managed code. It is now assumed that development of new .NET applications that don't require interop with native code will be much more likely to be done using C# or VB .NET.

About This Book

The purpose of this book is to show you the basics of the C++/CLI language. This book is not a general introduction to Visual C++. I'd like this book to be used as a handy desktop reference to the C++/CLI language, so if you have a question about how, say, an array is declared or how a ref class behaves, you can easily refer to this book. I am going to assume that you already have a working knowledge of C++, although the truth is that very few people know all there is to know about C++. However, I am assuming you know about as much as the majority of people who program in C++. I am assuming that you want to build on this existing knowledge, and may need the occasional refresher on the ins and outs of C++ as well. I do not assume any knowledge of the CLR, so if you have knowledge (perhaps from C# or Visual Basic .NET), you'll find a little bit of review. This book should be useful to professional developers getting started with C++/CLI, as well as to students, academic faculty, and hobbyists who want to learn the new language. In this text, I won't cover features of C++ that are not specifically C++/CLI extensions, even though C++/CLI does allow the use of nearly all of the C++ language. There are many good references available for classic C++[1].

Also, this book is an introductory book. There are many complexities that are not fully explained, especially in dealing with interoperability between native C++ and C++/CLI. If you want to move on to more advanced material after reading this book, you may want to read *Expert C++/CLI* by Marcus Heege (Apress, 2007), and if you want more information about using the .NET Framework in C++/CLI, you should read *Pro Visual C++/CLI and the .NET 2.0 Platform* by Stephen R.G. Fraser (Apress, 2006).

One of the principles with which this book is written is that, to paraphrase Einstein, explanations should be as simple as possible, but no simpler. I shall try to give many code examples that can be understood at a glance. I hope you won't need to spend a long time poring over the text and code in this book, but that you can absorb the main point of each code example and apply it to your own project. But, like any principle, there are times when it must be violated, so this book also contains more extended code examples that are intended to give you a better feeling for how the language is used in more realistic programs and get you thinking about how to solve problems using C++/CLI.

In Chapter 1, I introduce some of the basic concepts behind the new language, culminating in a look at the classic "Hello, World" program in C++/CLI. Following that, you'll get a quick

1. See, for example, *C++ Primer, Fourth Edition* by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo (Addison-Wesley, 2005) and *The C++ Programming Language, Special Third Edition* by Bjarne Stroustrup (Addison-Wesley, 2000).

tour of the new language in Chapter 2, using an example involving a simulation of radioactive decay to motivate the tour. You'll then look in Chapter 3 at some of the infrastructure outside of the programming language itself that you'll want to know about to program effectively in this environment, and in Chapter 4 you'll look at object semantics in the new language, as well as mixing native and managed objects. Chapter 5 covers the new C++/CLI features, starting with features of the CLI itself such as the `String` type and input/output, followed by enums and arrays. Chapter 6 describes classes and structs in C++/CLI. The text will then continue its treatment of classes in Chapter 7 by looking at new elements of a class, such as properties, events, and operators. Chapter 8 describes inheritance in C++/CLI. In Chapter 9, I discuss interface classes, which provide an alternative to traditional multiple inheritance. Chapter 10 describes other language features such as exception handling, which is the fundamental mechanism for error handling in the CLR; attributes, which provide metadata for a type; and reflection, the C++/CLI equivalent of runtime type information. Chapter 11 discusses parameterized types and collection classes, and Chapter 12, which is new for this edition, goes into the STL/CLR library. And finally, I round out your introduction to C++/CLI in Chapter 13 with a closer look at the features of the language supporting interoperability with existing native C++ code and other .NET languages. Throughout the text, I encourage you to experiment with the code examples and work on your own programs as well as those in the text. Example code can be found online at `http://www.apress.com`, and you can try out C++/CLI for yourself for free by downloading Visual C++ Express from `http://msdn.microsoft.com/vstudio/express`. I hope you learn much and enjoy reading this book.