SAMPLE CHAPTER

C++/CLI IN ACTION

Nishant Sivakumar

MANNING

C++/CLI in Action

by Nishant Sivakumar

Sample Chapter 1

brief contents

1 Introduction to C++/CLI

When C++ was wedded to CLI with a slash, it was apparent from the beginning that it wasn't going to be a celebrity marriage. The world's most powerful high level programming language—C++—was given a face-lift so that it could be used to develop on what could potentially be the world's most popular runtime environment: the CLI.

In this chapter, you'll see what C++/CLI can be used for and how C++/CLI improves the now-obsolete Managed C++ syntax. We'll also go over basic C++/CLI syntax. By the end of this chapter, you'll know how to write and compile a C++/CLI program and how to declare and use managed types. Some of the new syntactic features may take a little getting used to, but C++ as a language has never had simplicity as its primary design concern. Once you get used to it, you can harness the power and ingenuity of the language and put that to effective use.

1.1 The role of C++/CLI

C++ is a versatile programming language with a substantial number of features that makes it the most powerful and flexible coding tool for a professional developer. The Common Language Infrastructure (CLI) is an architecture that supports a dynamic language-independent programming model based on a Virtual Execution System. The most popular implementation of the CLI is Microsoft's .NET Framework for the Windows operating system. C++/CLI is a binding between the standard C++ programming language and the CLI. Figure 1.1 shows the relationship between standard C++ and the CLI.

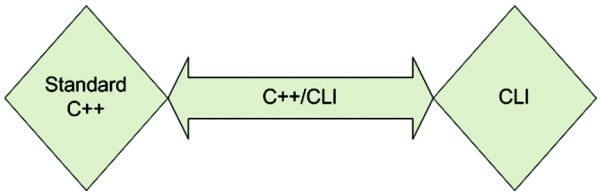

Figure 1.1
How C++/CLI connects standard C++ to the CLI

C++ has been paired with language extensions before, and the result hasn't always been pretty. Visual C++ 2005 is the first version of a Microsoft C++ compiler that has implemented the C++/CLI specification. This means three things for the C++ developer:

- C++ can be used to write applications that run on the .NET Framework. There is no need to learn a totally new language or to abandon all the C++ knowledge and experience built up through years of coding.

- C++/CLI lets developers reuse their native C++ code base, saving the agony of having to rewrite all the existing code to enable it to run on the .NET Framework.
- C++/CLI is designed to be the lowest-level language for the .NET Framework. For writing purely managed applications, it's your most powerful choice; or, as I like to say, "C++/CLI actually lets you smell the CLR (Common Language Runtime)."

Visual C++ 2005 isn't Microsoft's first attempt at providing a C++ compiler capable of targeting managed code. Both VC++ 2002 and VC++ 2003 featured a C++ compiler that supported the managed extensions to C++ (referred to as Managed C++ or MC++). As a syntactic extension, it would be an understatement to say that it was a comprehensive failure.

Now that you understand the role of C++/CLI, let's examine why it's such an invaluable inclusion among CLI languages.

The .NET Framework

It's important for you to have a basic understanding of the .NET Framework, because although this book will teach you the C++/CLI syntax before we move on to various interop mechanisms and strategies, it doesn't attempt to teach you the core details of the .NET Framework. If you've never used the .NET Framework, you should read the book's appendix ("A Concise Introduction to the .NET Framework") before you proceed further. On the other hand, if you've previously worked with the .NET Framework, you can refresh your memory by looking at these quick definitions (in no particular order) of various terms associated with the .NET Framework, which you'll encounter in this and other chapters:

- **.NET Framework**: The .NET Framework is Microsoft's implementation of the Common Language Infrastructure (CLI), which itself is an open specification that has been standardized by the ECMA (an international standards body). The .NET Framework consists of the Common Language Runtime (CLR) and the Base Class Library (BCL).
- **CLR**: The Common Language Runtime is the core of the .NET Framework and implements the fundamental aspects of the CLI such as the Virtual Execution System (VES), the Garbage Collector, and the Just in Time (JIT) compiler.
- **BCL**: The Base Class Library is an extensive set of .NET classes that is used by any .NET language (such as C#, VB.NET, C++/CLI, and so on).

- **VES**: The Virtual Execution System is the engine responsible for executing managed code, including the invocation of the Garbage Collector as well as the JIT compiler.
- **Garbage Collector**: Memory management is automatically done in the .NET Framework. The CLR includes a Garbage Collector that frees resources when they're no longer needed, so the developer doesn't need to worry about that.
- **JIT compiler**: .NET compilers (C#, VB.NET, C++/CLI, and so on) compile source code into an intermediate language called Microsoft Intermediate Language (MSIL). At runtime, the CLR uses the JIT compiler to compile this MSIL into the native code for the underlying operating system before executing it.
- **CTS**: The Common Type System (CTS) is a set of rules that specifies how the CLR can define, use, create, and manage types.
- **CLS**: The Common Language Specification (CLS) is a subset of the CTS that all languages must implement if they're to be considered CLS-compliant. CLS-compliant languages can interop with each other as long as they don't use any non-CLS-compliant features present in their specific compiler version.

I'd like to reiterate that if you're not familiar with these terms, or if you wish to understand them in more detail, please take a detour into the appendix, where most of these concepts are explained more thoroughly.

1.1.1 What C++/CLI can do for you

If you're reading this book, chances are good that you're looking to move your applications to the .NET Framework. The biggest concern C++ developers have about making the move to .NET is that they're afraid of abandoning their existing native code and having to rewrite everything to managed code. That's exactly where C++/CLI comes into the picture. You do *not* have to abandon your current native code, nor do you have to rewrite everything to managed code. That's C++/CLI's single biggest advantage—the ability to reuse existing native code.

Reuse existing native code

Visual C++ 2005 allows you to compile your entire native code base to MSIL with the flick of a single compilation switch. In practice, you may find that you have to change a small percentage of your code to successfully compile and build your applications. This is a far better option than either abandoning all your code or rewriting it entirely. Once you've successfully compiled your code for the CLR, the code can access the thousands of classes available in the .NET Base Class Library.

Access the entire .NET library

The .NET Framework comes with a colossal library containing thousands of classes that simplify your most common developmental requirements. There are classes relating to XML, cryptography, graphical user interfaces, database technologies, OS functionality, networking, text processing, and just about anything you can think of. Once you've taken your native applications and compiled them for the CLR, you can use these .NET classes directly from your code. For example, you can take a regular MFC dialog-based application and give it some encryption functionality using the .NET cryptography classes. You aren't restricted to managed libraries, because C++/CLI lets you seamlessly interop between managed and native code.

Most powerful language for interop

Although other languages like C# and VB.NET have interop features, C++/CLI offers the most powerful and convenient interop functionality of any CLI language. C++/CLI understands managed types as well as native types; consequently, you often end up using whatever library you want (whether it's a native DLL or a managed assembly) without having to worry about managed/native type conversions. Using a native library from C++/CLI is as simple as `#include`-ing the required header files, linking with the right lib files, and making your API or class calls as you would normally do. Compare that with C# or VB.NET, where you're forced to copy and paste numerous P/Invoke declarations before you can access native code. In short, for any sort of interop scenario, C++/CLI should be an automatic language choice. One popular use of interop is to access new managed frameworks such as Windows Forms from existing native applications. (Note that this technique is covered in detail in part 3 of the book.)

Leverage the latest managed frameworks

Imagine that you have a substantially large MFC application and that your company wants to give it a new look and feel. You've recently acquired an outstanding Windows Forms–based UI library from another company. Take VC++ 2005, recompile the MFC application for the CLR, and change the UI layer to use the Windows Forms library, and now you have the same application that uses the same underlying business logic with the new shiny user interface. You aren't restricted to Windows Forms or even to UI frameworks. The next version of Windows (called Windows Vista) will introduce a new UI framework called the Windows Presentation Foundation (WPF). It's a managed framework and C++/CLI will let you access it from existing native applications. When Vista is released, your

applications will be able to flaunt the WPF look and feel. Note that WPF is also being made available for Windows XP, so you aren't restricted to using Vista to run WPF-based applications.

Another powerful managed framework that is coming out in Vista is the Windows Communication Foundation (WCF), which, as the name implies, is a powerful communication framework written in managed code. And yes, although you knew I was going to say it, you can access the WCF from your Visual C++ applications. Although native code reuse and powerful interop are its most popular advantages, C++/CLI is also your most powerful option to write managed applications.

Write powerful managed applications

When Brandon Bray from the Visual C++ Compiler team said that C++/CLI would be the lowest-level language outside of MSIL, he meant what he said! C++/CLI supports more MSIL features than any other CLI language; it is to MSIL what C used to be to Assembly Language in the old days. C++/CLI is currently the only CLI language that supports stack semantics and deterministic destruction, mixed types, managed templates, and STL.NET (a managed implementation of the Standard Template Library).

A natural question you may have now is why Microsoft introduced a new syntax. Why didn't it continue to use the old MC++ syntax? That's what we examine next.

1.1.2 The rationale behind the new syntax

The managed extensions to C++ introduced in VC++ 2002 were not well received by the C++ developer community. Most people appreciated the fact that they could use C++ for .NET development, but almost everybody thought the syntax was gratuitously twisted and unnatural, that the managed and unmanaged pointer usage semantics were confusing, and that C++ hadn't been given equal footing as a CLI language with other languages like C# or VB.NET. Another factor that contributed to poor Managed C++ acceptance was the fact that designer support for Windows Forms was not available in the 2002 release; although the 2003 release did introduce a designer, it wasn't as stable or functional as the designers available for C# and VB.NET, and creating pure managed applications was dreadfully unfeasible with C++.

Microsoft took the feedback from its C++ developer community seriously, and on October 6, 2003, the ECMA (an association dedicated to the standardization of Information and Communication Technology and Consumer Electronics) announced the creation of a new task group to oversee the development of a standard set of language extensions to create a binding between the ISO

standard C++ programming language and the CLI. Microsoft developed and submitted full draft specifications for binding the C++ Programming Language to the Common Language Infrastructure in November 2003, and C++/CLI became an international ECMA standard in December 2005. It was expected that the ECMA would submit it to the ISO for consideration as a potential ISO standard. Visual C++ 2005 is the first publicly available compiler to support this new standard.

NOTE The ECMA (the acronym originally stood for European Computer Manufacturers Association) is an association founded in 1961 that's dedicated to the standardization of Information Technology systems. The ECMA has close liaisons with other technology standards organizations and is responsible for maintaining and publishing various standards documents. Note that the old acronym isn't used anymore, and the body today goes by the name ECMA International. You can visit its website at www.ecma-international.org.

Let's take a quick look at a few of the problems that existed in the old syntax and how C++/CLI improves on these issues. If you've used the old syntax in the past, you'll definitely appreciate the enhancements in the new syntax; if you haven't, you'll still notice the stark difference in elegance between the two syntaxes.

Twisted syntax and grammar

The old Managed C++ syntax used a lot of underscored keywords that were clunky and awkward. Note that these double-underscored keywords were required to conform to ANSI standards, which dictate that all compiler-specific keywords need to be prefixed with double underscores. But as a developer, you always want your code to feel natural and elegant. As long as developers felt that the code they wrote didn't look or feel like C++, they weren't going to feel comfortable using that syntax; most C++ developers chose not to use a syntax that felt awkward.

C++/CLI introduced a new syntax that fit in with existing C++ semantics. The elegant grammar gives a natural feel for C++ developers and allows a smooth transition from native coding to managed coding.

Look at table 1.1, which compares the old and new syntaxes; you'll see what I mean.

Even without knowing the rules for either the old or the new syntax or C++/CLI, you shouldn't find it hard to decide which is the more elegant and natural of the two. Don't worry if the code doesn't make a lot of sense to you right now. Later

Table 1.1 Comparison between old and new syntaxes

Old syntax	C++/CLI syntax

`__gc __interface I`
`{`
`};`
 	
`__delegate int ClickHandler();`
 	
`__gc class M : public I`
`{`
`  __event ClickHandler* OnClick;`
`public:`
`  __property int get_Num()`
`  {`
`  return 0;`
`  }`
`  __property void set_Num(int)`
`  {`
`  }`
`};`
	`interface class I`
	`{`
	`};`
	
	`delegate int ClickHandler();`
	
	`ref class M : public I`
	`{`
	`  event ClickHandler^ OnClick;`
	`public:`
	`  property int Num`
	`  {`
	`  int get()`
	`  {`
	`  return 0;`
	`  }`
	`  void set(int value)`
	`  {`
	`  }`
	`  }`
	`};`

in this chapter, we'll go through the fundamental syntactic concepts of the C++/CLI language. I just wanted to show you why the old syntax never became popular and how Microsoft has improved on the look and feel of the syntax in the new C++/CLI specification.

With the old syntax, every time you used a CLI feature, such as a delegate or a property, you had to prefix it with underscored keywords. For a property definition, the old syntax required separate setter and getter blocks and didn't syntactically organize them into a single block. This meant that if you carelessly separated the getter and setter methods with other code, there was no visual cue that they were part of the same property definition. With the new syntax, you put your getter and setter functions inside a property block; the relationship between them is visually maintained. To summarize my personal thoughts on this issue, with the old syntax, you feel that you're using two unrelated sublanguages (one for managed code and one for native code) with a single compiler. With the new syntax, you feel that you're using C++, albeit with a lot of new keywords: It's still a single language. Note that the VC++ team made an effort to ensure that the new keywords don't interfere with existing code bases, as you'll see later in the book.

Programmers can be compiler snobs. Many developers opined that C# and VB.NET were proper .NET languages, whereas MC++ was a second-class citizen compared to them. Let's see what has been done to the C++ language to promote it to a first-class CLI status.

Second class CLI support

Managed C++ seemed like a second-class CLI language when compared to languages like C# and VB.NET, and developers using it had to resort to contorted workarounds to implement CLI functionality. Take a trivial example such as enumerating over the contents of an `ArrayList` object. Here's what the code would look like in Managed C++:

```
IEnumerator* pEnumerator = arraylist->GetEnumerator();
while(pEnumerator->MoveNext())
{
    Console::WriteLine(pEnumerator->Current);
}
```

While languages like C# provided a `for each` construct that abstracted the entire enumeration process, C++ developers were forced to access the `IEnumerator` for the `ArrayList` object and use that directly—not a good thing, as far as Objected-Oriented abstraction rules were concerned. Now the programmer needs to know that the collection has an enumerator, that the enumerator has a `MoveNext` method and a `Current` property, and that they have to repeatedly call `MoveNext` until it returns `false`. This information should be hidden from the programmer. Requiring the internal implementation details of the collection to be directly used defeats the purpose of having collection classes, when the reason for having them is to abstract the internal details of an enumerable collection class from the programmer.

Look at the equivalent C++/CLI code:

```
for each(String^ s in arraylist)
{
    Console::WriteLine(s);
}
```

By adding constructs such as `for each`, which give developers a more natural syntax to access .NET features, the VC++ team has given us a cozy feeling that C++/CLI is now a first-class language for .NET programming. Later in this chapter, you'll see that boxing is now implicit, which means you don't have to use the gratuitous `__box` keyword required in the old syntax. If you don't know what boxing means, don't worry; it will be explained when we talk about boxing and unboxing.

Poor integration of C++ and .NET

One major complaint about Managed C++ was that C++ features such as templates and deterministic destruction weren't available. Most C++ developers felt severely handicapped by the apparent feature reductions when using MC++.

With C++/CLI, templates are supported on both managed and unmanaged types. In addition, C++/CLI is the only CLI language that supports stack semantics and deterministic destruction (although languages like C# 2.0 use indirect workarounds like the `using-block` construct to conjure up a form of deterministic destruction).

A crisp summarization would be to say that C++/CLI bridges the gap between C++ and .NET by bringing C++ features such as templates and deterministic destruction to .NET, and .NET features like properties, delegates, garbage collection, and generics to C++.

Confusing pointer usage

Managed C++ used the same `*` punctuator-based operator syntax for unmanaged pointers into the C++ heap and managed references into the CLI heap. Not only was this confusing and error-prone, but managed references were different entities with totally different behavioral patterns from unmanaged pointers. Consider the following code snippet:

```
__gc class R
{
};

class N
{
};

. . .

N* pN = new N();
R* pR = new R();
```

The two calls to `new` (shown in bold) do completely different things. The `new` call on the native class `N` results in the C++ `new` operator being called, whereas the `new` call on the managed class `R` is compiled into the MSIL `newobj` instruction. The native object is allocated on the C++ heap, but the managed object is allocated on the garbage-collected CLR heap, which has the side implication that the memory address for the object may change every time there is a garbage-collection cycle or a heap-compaction operation. The `R*` object looks like a native C++ pointer, but it doesn't behave like one, and its address can't be assumed to remain

fixed. The good news is that this issue has been fixed in C++/CLI. We now have an additional `gcnew` keyword for instantiating managed objects. We also have the concept of a *handle* (as opposed to a pointer) to a managed object that uses the ^ punctuator (instead of *) as the handle operator. Later in this chapter, we'll take a more detailed look at handles and the `gcnew` operator. For now, it should suffice to note that in C++/CLI, there will be no managed/unmanaged pointer confusion.

Unverifiable code

The Managed C++ compiler could not produce verifiable code, which meant that you couldn't use it to write code that was to run under a protected environment—such as an SQL Server-stored procedure. Visual C++ 2005 supports a special compiler mode (`/clr:safe`) that produces verifiable code and disallows you from compiling any non-verifiable code by generating errors during compilation. The advantage of being able to create verifiable assemblies is that the CLR can enforce active CLR security restrictions on the running application. This gives you a wider scope to deploy your applications (for example, as SQL Server components) in secure environments like those in a banking system and in future Windows releases where code may have to be verifiable to be permitted to execute.

It should be obvious by now why Microsoft decided to bring out a new syntax. If you've never used the old syntax, you can consider yourself lucky that you can now use the powerful new C++/CLI language to write managed applications.

If you *have* used the old syntax, I strongly recommend spending some time porting the old syntax code to the new syntax as early as possible. The old syntax support (available in VC++ 2005 through the `/clr:oldSyntax` compiler switch) isn't guaranteed to be available in future VC++ versions, nor will any significant improvements be made to it. Let's now move on to our first C++/CLI program.

1.2 Hello World in C++/CLI

Before we go any further, let's write our first Hello World application in C++/CLI. In this section, we'll also look at the new compiler options that have been introduced in VC++ 2005 to support compilation for managed code. There is nothing overly-complicated about the code in Listing 1.1, but for our purposes, it does nicely to illustrate a few basic language concepts of C++/CLI.

Listing 1.1 Hello World program in C++/CLI

```
#pragma comment(lib, "Advapi32")
#include <windows.h>
#include <tchar.h>
```

```
#include <lmcons.h>
using namespace System;

int main()
{
    TCHAR buffer[UNLEN + 1];
    DWORD size = UNLEN + 1;
    GetUserName(buffer, &size);
    String^ greeting = "Hello";
    Console::WriteLine("{0} {1}",
        greeting, gcnew String(buffer));
    return 0;
}
```

❶ Get current user

❷ Display greeting

You can compile this code from the command line using the C++ compiler cl.exe as follows:

```
cl /clr First.cpp
```

Run it, and it promptly displays "Hello" followed by the current user (most likely your Windows login name) on the console. Except for the `gcnew` keyword (which we'll talk about later in this chapter), it doesn't look very different from a regular C++ program, does it? But the executable that has been created is a .NET executable that runs on the .NET Common Language Runtime. When I say *.NET executable*, I mean an MSIL program that is JIT compiled and executed by the CLR just like any executable you might create using C#, VB.NET, or another CLI language. This is a small example, but it communicates two important facts: You can use familiar C++ syntax to write .NET applications, thereby avoiding the need to learn a new language like C# or VB.NET; and you can write managed ❷ and native code ❶ within the same application.

For those of you who aren't familiar with the .NET Framework, `Console` is a .NET Framework BCL class that belongs to the `System` namespace (hence the `using namespace` declaration on top), and `WriteLine` is a static method of the `Console` class. Listing 1.1 uses native data types like `TCHAR` and `DWORD` as well as managed data types like `String`. Similarly, it uses a native Win32 API call (`GetUserName`) as well as a managed class (`System::Console`). The best part is that you do all this in a single application (within a single function, in this case). Although you may not have realized it, you've just written a mixed-mode application that mixes native and managed code. Congratulations! You can do a lot with mixed-mode coding, and you'll see far more useful applications of that technique throughout the later portions of this book.

You must have observed that I specified `/clr` as a compiler option. Let's talk a little more about that.

1.2.1 The /clr compiler option

To use the C++/CLI language features, you need to enable the `/clr` compiler switch; without it, cl.exe behaves like a native C++ compiler. The `/clr` switch creates a .NET application that's capable of consuming .NET libraries and can take advantage of CLR features such as managed types and garbage collection. You can specify suboptions to the `/clr` option to further specify the type of assembly you want created. Table 1.2 is a partial list of the `/clr` suboptions you can specify and what they do. For a more complete list, refer to the MSDN documentation for the C++ compiler command-line switches.

Now that we've discussed the command-line compiler options, let's look at how you can use the VC++ 2005 environment to create C++/CLI projects.

Table 1.2 Partial listing of `/clr` compilation modes in VC++ 2005

Compiler switch	Description
`/clr`	Creates an assembly targeting the CLR. The output file may contain both MSIL and native code (mixed-mode assemblies). This is the most commonly-used switch (which is probably why it's the default). It lets you enable CLR support to native C++ projects including, but not limited to, projects that use MFC, ATL, WTL, STL, and Win32 API. This will be the most commonly-used compilation mode throughout this book.
`/clr:pure`	Creates an MSIL-only assembly with no native code (hence *pure*). You can have native (unmanaged) types in your code as long as they can be compiled into pure MSIL. C# developers can think of this as being equivalent to using the C# compiler in unsafe mode—the output is pure MSIL but not necessarily verifiable.
`/clr:safe`	Creates an MSIL-only verifiable assembly. You can't have native types in your code, and if you try to use them, the compiler will throw an error. This compilation mode produces assemblies that are equivalent to what C# (regular mode) and VB.NET would produce.
`/clr:oldSyntax`	Enables the MC++ syntax available in VC++ 2002 and VC++ 2003. I strongly advocate that you never use this option, except where it's an absolute necessity. Even if it takes considerable time to port a large old syntax code base to the new syntax, it's still your best option in the long run. There is no guarantee that this option will be available in a future version of the VC++ compiler.

1.2.2 Using VC++ 2005 to create a /clr application

For any nontrivial program, it makes sense to use the Visual C++ development environment, although it's still good to know the compiler options available. I believe that one of the biggest reasons for the popularity of the VC++ compiler is the fact that it comes with a powerful development environment, and there's no reason we shouldn't take advantage of it. For the rest of this chapter and the next two chapters, we'll use CLR-enabled console applications as we look at the C++/CLI syntax and grammar. Those of you who want to follow along in your own console project can type in the code as it's written in the book.

NOTE In later chapters, where the examples are longer and more complex, you can use the book's companion CD, which contains full source code for the samples.

Creating a CLR console application with Visual C++ is straightforward. The steps are as follows.

1. In the New Project Wizard dialog, choose *CLR* under *Visual C++* in the *Project types* tree control on the left, and select *CLR Console Application* from the *Templates* list control on the right. You can use Figure 1.2 as a reference when doing this.
2. Enter a name for the project, and click OK.

The wizard generates quite a few files for you. The one that should interest you most is the CPP file that has the same name as the project. If you named your project Chapter01Demo, you'll see a Chapter01Demo.cpp file in your solution that contains the wizard-generated `main` method. You must have used similar wizards in the past when working on MFC, ATL, or Win32 API projects, so this should be familiar.

You'll notice something interesting about the way the generated `main` function is prototyped:

```
int main(array<System::String ^> ^args)
```

This version of `main` is compatible with the entry-point prototypes available for C# and VB.NET programs and adheres to the CLI definition of a managed entry-point function. The syntax may seem a little confusing right now (because we haven't yet begun exploring the C++/CLI syntax), but `args` is essentially a managed array of `System::String` objects that represents the command-line arguments passed to the application. Keep this important distinction in mind: Unlike

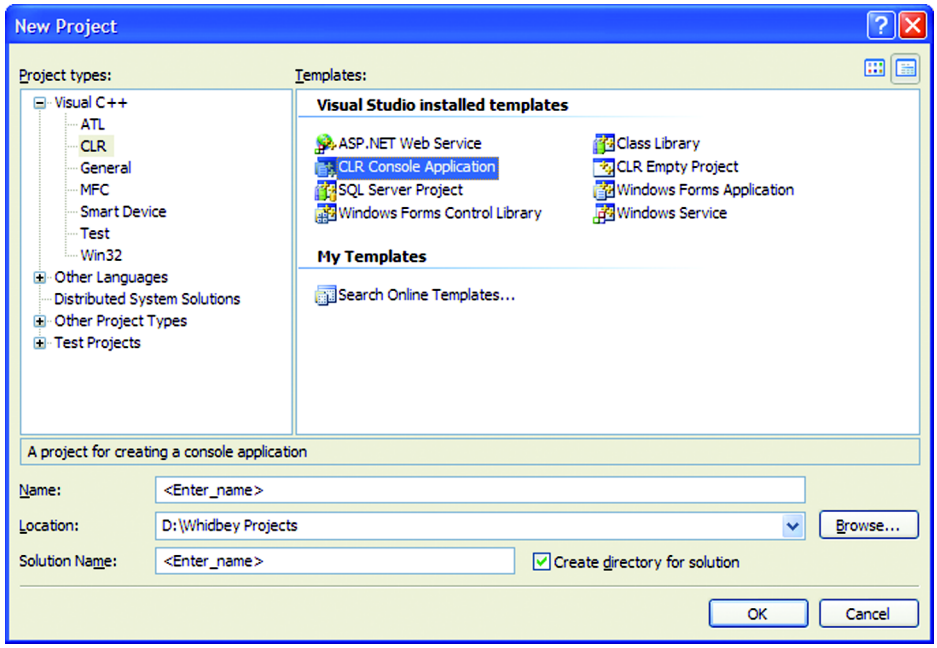

Figure 1.2 The Visual C++ 2005 New Project Wizard

the native C++ `main` prototypes, the name of the program isn't passed as the zero-indexed argument. If you run the application without any command-line arguments, the array will be empty. By default, the wizard sets the project to use the `/clr` compilation option, but you can change that by using the Project Properties dialog; you can access it by choosing Project > Properties or by using the Alt-F7 keyboard shortcut.

NOTE Your keyboard shortcuts will vary depending on your VS profile. This book uses the shortcuts associated with the default VC++ profile.

Select *General* from *Configuration Properties* on the left, and you'll see an option to set the `/clr` compilation switch (you can choose from `/clr`, `/clr:pure`, `/clr:safe`, and `/clr:oldSyntax`), as shown in Figure 1.3.

Now that you've seen how to create a C++/CLI project, let's look at the type-declaration syntax for declaring CLI types (also referred to as CLR types). When you learn how to declare and use CLR types, you get a proper feel for programming on top of the CLR.

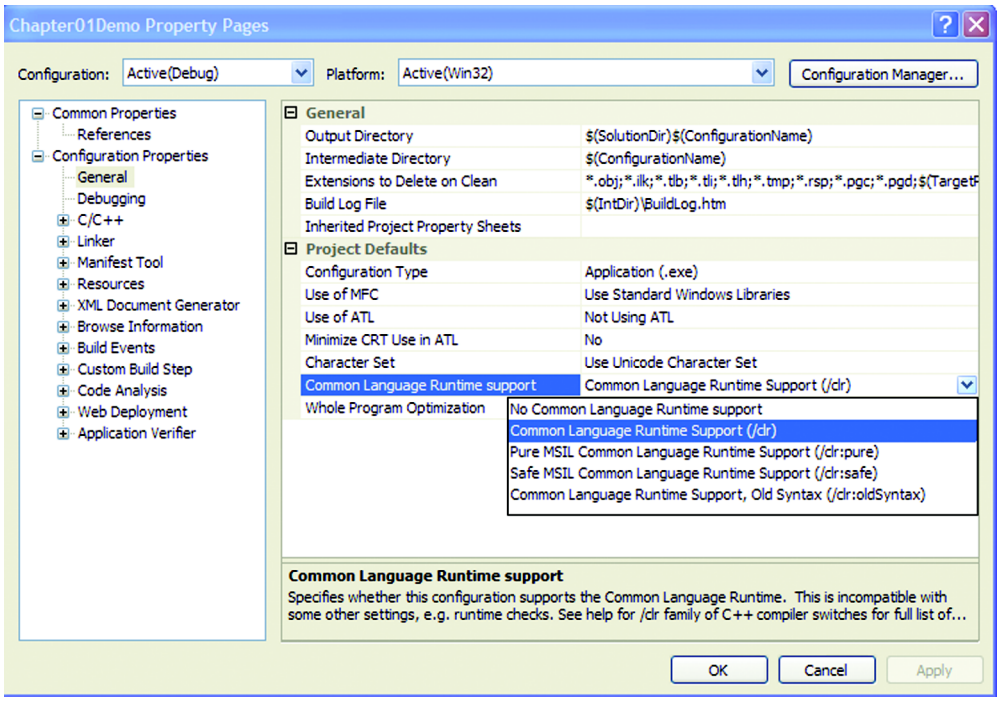

Figure 1.3 Setting the CLR compilation options using the Project Properties dialog

1.3 Declaring CLR types

In this section, we'll look at the syntax for declaring CLI (or CLR) types, modifiers that can be applied to CLI types, and how CLI types implement inheritance. C++/CLI supports both native (unmanaged) and managed types and uses a consistent syntax for declaring various types. Native types are declared and used just as they are in standard C++. Declaring a CLI type is similar to declaring a native type, except that an adjective is prefixed to the class declaration that indicates the type being declared. Table 1.3 shows examples of CLI type declarations for various types.

C# developers may be a little confused by the usage of both `class` and `struct` for both reference and value types. In C++/CLI, `struct` and `class` can be used interchangeably (just as in standard C++), and they follow standard C++ visibility rules for structs and classes. In a class, methods are `private` by default; in a struct, methods are `public` by default. In table 1.3, `RefClass1::Func` and `ValClass1::Func` are both `private`, whereas `RefClass2::Func` and `ValClass2::Func`

Table 1.3 Type declaration syntax for CLI types

CLI type	Declaration syntax

Reference types	`ref class RefClass1`
	`{`
	`  void Func(){}`
	`};`
	
	`ref struct RefClass2`
	`{`
	`  void Func(){}`
	`};`

Value types	`value class ValClass1`
	`{`
	`  void Func(){}`
	`};`
	
	`value struct ValClass2`
	`{`
	`  void Func(){}`
	`};`

Interface types	`interface class IType1`
	`{`
	`  void Func();`
	`};`
	
	`interface struct IType2`
	`{`
	`  void Func();`
	`};`

are both `public`. For the sake of clarity and consistency with C#, you may want to exclusively use `ref class` for ref types and `value struct` for value types instead of mixing `class` and `struct` for both ref and value types.

Interface methods are always `public`; declaring an interface as a struct is equivalent to declaring it as a class. This means `IType1::Func` and `IType2::Func` are both `public` in the generated MSIL. C# developers must keep the following in mind:

- A C++/CLI value class (or value struct) is the same as a C# struct.
- A C++/CLI ref class (or ref struct) is the same as a C# class.

Those of you who have worked on the old MC++ syntax should remember these three points:

- A ref class is the same as an `__gc` class.
- A value class is the same as an `__value` class.
- An interface class is the same as an `__interface`.

Spaced keywords

An interesting thing that you need to be aware of is that only three new, reserved keywords have been introduced in C++/CLI: `gcnew`, `nullptr`, and `generic`. All the other seemingly new keywords are *spaced* (or contextual) keywords. Syntactic phrases like `ref class`, `for each`, and `value class` are spaced keywords that are treated as single tokens in the compiler's lexical analyzer. The big advantage is that any existing code that uses these new keywords (like `ref` or `each`) continues to compile correctly, because it's not legal in C++ to use a space in an identifier. The following code is perfectly valid in C++/CLI:

```
int ref = 0;
int value = ref;
bool each = value == ref;
```

Of course, if your existing code uses `gcnew`, `nullptr`, or `generic` as an identifier, C++/CLI won't compile it, and you'll have to rename those identifiers.

You've seen how CLI types can be declared. Next, you'll see how type modifiers can be applied to these classes (or structs, as the case may be).

1.3.1 Class modifiers

You can specify the `abstract` and `sealed` modifiers on classes; a class can be marked both `abstract` and `sealed`. But such classes can't be derived explicitly from any base class and can only contain `static` members. Because global functions aren't CLS-compliant, you should use abstract sealed classes with `static` functions, instead of global functions, if you want your code to be CLS-compliant.

In case you're wondering when and why you would need to use these modifiers, remember that, to effectively write code targeting the .NET Framework, you should be able to implement every supported CLI paradigm. The CLI explicitly supports abstract classes, sealed classes, and classes that are both abstract and sealed. If the CLI supports it, you should be able to do so, too.

Just as with standard C++, an abstract class can only be used as a base class for other classes. It isn't required that the class contains abstract methods for it to be declared as an abstract class, which gives you extra flexibility when designing your

class hierarchy. The following class is abstract because it's declared `abstract`, although it doesn't contain any abstract methods:

```
ref class R2 abstract
{
public:
    virtual void Func(){}
};
```

An interesting compiler behavior is that if you have a class with an abstract method that isn't marked `abstract`, such as the following class, the compiler issues warning C4570 (*class is not explicitly declared as abstract but has abstract functions*) instead of issuing an error:

```
ref class R1
{
public:
    virtual void Func() abstract;
};
```

In the generated IL, the class R1 is marked `abstract`, which means that if you try to instantiate the class, you'll get a compiler error (and you should). Not marking a class `abstract` when it has abstract methods is untidy, and I strongly encourage you to explicitly mark classes `abstract` if at least one of their methods is abstract. Note how I've used the `abstract` modifier on a class method in the previous example; you'll see more on this and other function modifiers in chapter 2.

Using the `sealed` modifier follows a similar syntax. A sealed class can't be used as a base class for any other class—it seals the class from further derivation:

```
ref class S sealed
{
};

ref class D : S    // This won't compile
{                  // Error C3246
};
```

Sealed classes are typically used when you don't want the characteristics of a specific class to be modified (through a derived class), because you want to ensure that all instances of that class behave in a fixed manner. Because a derived class can be used anywhere the base class can be used, if you allow your class to be inherited from, by using instances of the derived class where the base class instance is expected, users of your code can alter the expected functionality (which you want to remain unchangeable) of the class. For example, consider a banking application that has a `CreditCardInfo` class that is used to fetch information about an account

holder's credit-card transactions. Because instances of this class will be occasionally transmitted across the Internet, all internal data is securely stored using a strong encryption algorithm. By allowing the class to be inherited from, there is the risk of an injudicious programmer forgetting to properly follow the data encryption implemented by the `CreditCardInfo` class; thus any instance of the derived class is inherently insecure. By marking the `CreditCardInfo` class as `sealed`, such a contingency can be easily avoided.

A performance benefit of using a sealed class is that, because the compiler knows a sealed class can't have any derived classes, it can statically resolve virtual member invocations on a sealed class instance using nonvirtual invocations. For example, assuming that the `CreditCardInfo` class overrides the `GetHashCode` method (which it inherits from `Object`), when you call `GetHashCode` at runtime, the CLR doesn't have to figure out which function to call. This is the case because it doesn't have to determine the polymorphic type of the class (because a `CreditCardInfo` object can only be a `CreditCardInfo` object, it can't be an object of a derived type—there are no derived types). It directly calls the `GetHashCode` method defined by the `CreditCardInfo` class.

Look at the following example of an abstract sealed class:

```
ref class SA abstract sealed
{
public:
       static void DoStuff(){}
private:
       static int bNumber = 0;
};
```

As mentioned earlier, abstract sealed classes can't have instance methods; attempting to include them will throw compiler error C4693. This isn't puzzling when you consider that an instance method on an abstract sealed class would be worthless, because you can never have an instance of such a class. An abstract sealed class can't be explicitly derived from a base class, although it implicitly derives from `System::Object`. For those of you who've used C#, it may be interesting to know that an abstract sealed class is the same as a C# static class.

Now that we've discussed how to declare CLI types and apply modifiers on them, let's look at how CLI types work with inheritance.

1.3.2 CLI types and inheritance

Inheritance rules are similar to those in standard C++, but there are differences, and it's important to realize what they are when using C++/CLI. The good thing is

that most of the differences are obvious and natural ones dictated by the nature of the CLI. Consequently, you won't find it particularly strenuous to remember them.

Reference types (`ref class`/`struct`) only support `public` inheritance, and if you skip the `access` keyword, `public` inheritance is assumed:

```
ref class Base
{
};

ref class Derived : Base // implicitly public
{
};
```

If you attempt to use `private` or `protected` inheritance, you'll get compiler error C3628. The same rule applies when you implement an interface; interfaces must be implemented using `public` inheritance, and if you skip the `access` keyword, `public` is assumed:

```
interface class IBase
{
};

ref class Derived1 : private IBase {}; //error C3141
ref class Derived2 : protected IBase {}; //error C3141
ref class Derived3 : IBase {}; //public assumed
```

The rules for value types and inheritance are slightly different from those for ref types. A value type can only implement interfaces; it can't inherit from another value or ref type. That's because value types are implicitly derived from `System::ValueType`. Because CLI types don't support multiple base classes, value types can't have any other base class. In addition, value types are always sealed and can't be used as base classes. In the following code snippet, only the `Derived3` class compiles. The other two classes attempt to inherit from a ref class and a value class, neither of which is permitted:

```
ref class RefBase {};
value class ValBase {};
interface class IBase {};

value class Derived1 : RefBase {}; //error C3830
value class Derived2 : ValBase {}; //error C3830
value class Derived3 : IBase {};
```

These restrictions are placed on value types because value types are intended to be simple types without the complexities of inheritance or referential identity, which can be implemented using basic copy-by-value semantics. Also note that

these restrictions are imposed by the CLI and not by the C++ compiler. The C++ compiler merely complies with the CLI rules for value types. As a developer, you need to keep these restrictions in mind when designing your types. Value types are kept simple to allow the CLR to optimize them at runtime where they're treated like simple plain old data (POD) types like an `int` or a `char`, thus making them extremely efficient compared to reference types.

Here's a simple rule you can follow when you want to decide whether a class should be a value type: Try to determine if you want it to be treated as a class or as plain data. If you want it to be treated as a class, don't make it a value type; but if you want it to behave just as an `int` or a `char` would, chances are good that your best option is to declare it as a value type. Typically, you'll want it to be treated as a class if you expect it to support virtual methods, user-defined constructors, and other aspects characteristic of a complex data type. On the other hand, if it's just a class or a struct with some data members that are themselves value types, such as an `int` or `char`, you may want to make that a value type.

One important point to be aware of is that CLI types don't support multiple inheritance. So, although a CLI type can implement any number of interfaces, it can have only one immediate parent type; if none is specified, this is implicitly assumed to be `System::Object`.

Next, we'll talk about one of the most important features that have been introduced in VC++ 2005: the concept of handles.

1.4 Handles: the CLI equivalent to pointers

Handles are a new concept introduced in C++/CLI; they replace the `__gc` pointer concept used in Managed C++. Earlier in the chapter, we discussed the pointer-usage confusion that prevailed in the old syntax. Handles solve that confusion. In my opinion, the concept of handles has contributed the most in escalating C++ as a first-class citizen of the .NET programming language world. In this section, we'll look at the syntax for using handles. We'll also cover the related topic of using tracking references.

1.4.1 Syntax for using handles

A *handle* is a reference to a managed object on the CLI heap and is represented by the ^ punctuator (pronounced *hat*).

NOTE When I say *punctuator* in this chapter, I'm talking from a compiler perspective. As far as the language syntax is concerned, you can replace the word *punctuator* with *operator* and retain the same meaning.

Handles are to the CLI heap what native pointers are to the native C++ heap; and just as you use pointers with heap-allocated native objects, you use handles with managed objects allocated on the CLI heap. Be aware that although native pointers need not always necessarily point to the native heap (you could get a native pointer pointing to the managed heap or to non-C++ allocated memory storage), managed handles have a close-knit relationship with the managed heap. The following code snippet shows how handles can be declared and used:

```
String^ str = "Hello world";
Student^ student = Class::GetStudent("Nish");
student->SelectSubject(150);
```

In the code, `str` is a handle to a `System::String` object on the CLI heap, `student` is a handle to a `Student` object, and `SelectSubject` invokes a method on the `student` handle.

The memory address that `str` refers to isn't guaranteed to remain constant. The `String` object may be moved around after a garbage-collection cycle, but `str` will continue to be a reference to the same `System::String` object (unless it's programmatically changed). This ability of a handle to change its internal memory address when the object it has a reference to is moved around on the CLI heap is called *tracking*.

Handles may look deceitfully similar to pointers, but they are totally different entities when it comes to behavior. Table 1.4 illustrates the differences between handles and pointers.

Table 1.4 Differences between handles and pointers

Handles	Pointers
Handles are denoted by the ^ punctuator.	Pointers are denoted by the * punctuator.
Handles are references to managed objects on the CLI heap.	Pointers point to memory addresses.
Handles may refer to different memory locations throughout their lifetime, depending on GC cycles and heap compactions.	Pointers are stable, and garbage-collection cycles don't affect them.
Handles track objects, so if the object is moved around, the handle still has a reference to that object.	If an object pointed to by a native pointer is programmatically moved around, the pointer isn't updated.
Handles are type-safe.	Pointers weren't designed for type-safety.

continued on next page

Table 1.4 Differences between handles and pointers *(continued)*

Handles	Pointers
The `gcnew` operator returns a handle to the instantiated CLI object.	The `new` operator returns a pointer to the instantiated native object on the native heap.
It isn't mandatory to delete handles. The Garbage Collector eventually cleans up all orphaned managed objects.	It's your responsibility to call `delete` on pointers to objects that you've allocated; if you don't do so, you'll suffer a memory leak.
Handles can't be converted to and from a `void^`.	Pointers can convert to and from a `void*`.
Handles don't allow handle arithmetic.	Pointer arithmetic is a popular mechanism to manipulate native data, especially arrays.

Despite all those differences, typically you'll find that for most purposes, you'll end up using handles much the same way you would use pointers. In fact, the `*` and `->` operators are used to dereference a handle (just as with a pointer). But it's important to be aware of the differences between handles and pointers. The VC++ team members initially called them managed pointers, GC pointers, and tracking pointers. Eventually, the team decided to call them handles to avoid confusion with pointers; in my opinion, that was a smart decision.

Now that we've covered handles, it's time to introduce the associated concept of tracking references.

1.4.2 Tracking references

Just as standard C++ supports references (using the `&` punctuator) to complement pointers, C++/CLI supports tracking references that use the `%` punctuator to complement handles. The standard C++ reference obviously can't be used with a managed object on the CLR heap, because it's not guaranteed to remain in the same memory address for any period of time. The tracking reference had to be introduced; and, as the name suggests, it tracks a managed object on the CLR heap. Even if the object is moved around by the GC, the tracking reference will still hold a reference to it. Just as a native reference can bind to an l-value, a tracking reference can bind to a managed l-value. And interestingly, by virtue of the fact that an l-value implicitly converts to a managed l-value, a tracking reference can bind to native pointers and class types, too. Let's look at a function that accepts a `String^` argument and then assigns a string to it. The first version doesn't work as expected; the calling code finds that the `String` object it passed to the function hasn't been changed:

```
void ChangeString(String^ str)
{
    str = "New string";
}
int main(array<System::String^>^ args)
{
    String^ str = "Old string";
    ChangeString(str);
    Console::WriteLine(str);
}
```

If you execute this code snippet, you'll see that `str` contains the old string after the call to `ChangeString`. Change `ChangeString` to

```
void ChangeString(String^% str)
{
    str = "New string";
}
```

You'll now see that `str` does get changed, because the function takes a tracking reference to a handle to a `String` object instead of a `String` object, as in the previous case. A generic definition would be to say that for any type `T`, `T%` is a tracking reference to type `T`. C# developers may be interested to know that MSIL-wise, this is equivalent to passing the `String` as a C# `ref` argument to `ChangeString`. Therefore, whenever you want to pass a CLI handle to a function, and you expect the handle itself to be changed within the function, you need to pass a tracking reference to the handle to the function.

In standard C++, in addition to its use in denoting a reference, the `&` symbol is also used as a unary address-of operator. To keep things uniform, in C++/CLI, the unary `%` operator returns a handle to its operand, such that the type of `%T` is `T^` (handle to type `T`). If you plan to use stack semantics (which we'll discuss in the next chapter), you'll find yourself applying the unary `%` operator quite a bit when you access the .NET Framework libraries. This is because the .NET libraries always expect a handle to an object (because C++ is the only language that supports a nonhandle reference type); so, if you have an object declared using stack semantics, you can apply the unary `%` operator on it to get a handle type that you can pass to the library function. Here's some code showing how to use the unary `%` operator:

```
Student^ s1 = gcnew Student();
Student% s2 = *s1;  // Dereference s1 and assign
                    // to the tracking reference s2
Student^ s3 = %s2;  // Apply unary % on s2 to return a Student^
```

Be aware that the `*` punctuator is used to dereference both pointers and handles, although symmetrically thinking, a `^` punctuator should been used to dereference

a handle. Perhaps this was designed this way to allow us to write agnostic template/generic classes that work on both native and unmanaged types.

You now know how to declare a CLI type; you also know how to use handles to a CLI type. To put these skills to use, you must understand how CLI types are instantiated, which is what we'll discuss in the next section.

1.5 Instantiating CLI classes

In this section, you'll see how CLI classes are instantiated using the `gcnew` operator. You'll also learn how constructors, copy constructors, and assignment operators work with managed types. Although the basic concepts remain the same, the nature of the CLI imposes some behavioral differences in the way constructors and assignment operators work; when you start writing managed classes and libraries, it's important that you understand those differences. Don't worry about it, though. Once you've seen how managed objects work with constructors and assignment operators, the differences between instantiating managed and native objects will automatically become clear.

1.5.1 The gcnew operator

The `gcnew` operator is used to instantiate CLI objects. It returns a handle to the newly created object on the CLR heap. Although it's similar to the `new` operator, there are some important differences: `gcnew` has neither an array form nor a placement form, and it can't be overloaded either globally or specifically to a class. A placement form wouldn't make a lot of sense for a CLI type, when you consider that the memory is allocated by the Garbage Collector. It's for the same reason you aren't permitted to overload the `gcnew` operator. There is no array form for `gcnew` because CLI arrays use an entirely different syntax from native arrays, which we'll cover in detail in the next chapter. If the CLR can't allocate enough memory for creating the object, a `System::OutOfMemoryException` is thrown, although chances are slim that you'll ever run into that situation. (If you do get an `OutOfMemoryException`, and your system isn't running low on virtual memory, it's likely due to badly written code such as an infinite loop that keeps creating objects that are erroneously kept alive.) The following code listing shows a typical usage of the `gcnew` keyword to instantiate a managed object (in this case, the `Student` object):

```
ref class Student
{
...
```

```
};
...

Student^ student = gcnew Student();
student->SelectSubject("Math", 97);
```

The gcnew operator is compiled into the newobj MSIL instruction by the C++/CLI compiler. The newobj MSIL instruction creates a new CLI object—either a ref object on the CLR heap or a value object on the stack—although the C++/CLI compiler uses a different mechanism to handle the usage of the gcnew operator to create value type objects (which I'll describe later in this section). Because gcnew in C++ translates to newobj in the MSIL, the behavior of gcnew is pretty much dependent on, and therefore similar to, that of the newobj MSIL instruction. In fact, newobj throws System::OutOfMemoryException when it can't find enough memory to allocate the requested object. Once the object has been allocated on the CLR heap, the constructor is called on this object with zero or more arguments (depending on the constructor overload that was used). On successful completion of the call to the constructor, gcnew returns a handle to the instantiated object. It's important to note that if the constructor call doesn't successfully complete, as would be the case if an exception was raised inside the constructor, gcnew won't return a handle. This can be easily verified with the following code snippet:

```
ref class Student
{
public:
    Student()
    {
        throw gcnew Exception("hello world");
    }
};

//...

Student^ student = nullptr; //initialize the handle to nullptr

try
{
    student =  gcnew Student(); //attempt to create object
}
catch(Exception^)
{
}

if(student == nullptr) //check to see if student is still nullptr
    Console::WriteLine("reference not allocated to handle");
```

Not surprisingly, `student` is still `nullptr` when it executes the `if` block. Because the constructor didn't complete executing, the CLR concludes that the object hasn't fully initialized, and it doesn't push the handle reference on the stack (as it would if the constructor had completed successfully).

NOTE C++/CLI introduces the concept of a universal null literal called `nullptr`. This lets you use the same literal (`nullptr`) to represent a null pointer and a null handle value. The `nullptr` implicitly converts to a pointer or handle type; for the pointer, it evaluates to 0, as dictated by standard C++; for the handle, it evaluates to a null reference. You can use the `nullptr` in relational, equality, and assignment expressions with both pointers and handles.

As I mentioned earlier, using `gcnew` to instantiate a value type object generates MSIL that is different from what is generated when you instantiate a ref type. For example, consider the following code, which uses `gcnew` to instantiate a value type:

```
value class Marks
{
public:
    int Math;
    int Physics;
    int Chemistry;
};

//...

    Marks^ marks = gcnew Marks();
```

For this code, the C++/CLI compiler uses the `initobj` MSIL instruction to create a `Marks` object on the stack. This object is then boxed to a `Marks^` object. We'll discuss boxing and unboxing in the next section; for now, note that unless it's imperative to the context of your code to `gcnew` a value type object, doing so is inefficient. A stack object has to be created, and this must be boxed to a reference object. Not only do you end up creating two objects (one on the managed stack, the other on the managed heap), but you also incur the cost of boxing. The more efficient way to create an object of type `Marks` (or any value type) is to declare it on the stack, as follows:

```
Marks marks;
```

You've seen how calling `gcnew` calls the constructor on the instance of the type being created. In the coming section, we'll take a more involved look at how constructors work with CLI types.

1.5.2 Constructors

If you have a ref class, and you haven't written a default constructor, the compiler generates one for you. In MSIL, the constructor is a specially-named instance method called `.ctor`. The default constructor that is generated for you calls the constructor of the immediate base class for the current class. If you haven't specified a base class, it calls the `System::Object` constructor, because every ref object implicitly derives from `System::Object`. For example, consider the following two classes, neither of which has a user-defined constructor:

```
ref class StudentBase
{
};
ref class Student: StudentBase
{
};
```

Neither `Student` nor `StudentBase` has a user-provided default constructor, but the compiler generates constructors for them. You can use a tool such as ildasm.exe (the IL Disassembler that comes with the .NET Framework) to examine the generated MSIL. If you do that, you'll observe that the generated constructor for `Student` calls the constructor for the `StudentBase` object:

```
call instance void StudentBase::.ctor()
```

The generated constructor for `StudentBase` calls the `System::Object` constructor:

```
call instance void [mscorlib]System.Object::.ctor()
```

Just as with standard C++, if you have a constructor—either a default constructor or one that takes one or more arguments—the compiler won't generate a default constructor for you. In addition to instance constructors, ref classes also support static constructors (not available in standard C++). A static constructor, if present, initializes the static members of a class. Static constructors can't have parameters, must also be `private`, and are automatically called by the CLR. In MSIL, static constructors are represented by a specially named static method called `.cctor`. One possible reason both special methods have a `.` in their names is that this avoids name clashes, because none of the CLI languages allow a `.` in a function name. If you have at least one static field in your class, the compiler generates a default static constructor for you if you don't include one on your own. When you have a simple class, such as the following, the generated MSIL will have a static constructor even though you haven't specified one:

```
ref class StudentBase
{
```

```
    static int number;
};
```

Due to the compiler-generated constructors and the implicit derivation from `System::Object`, the generated class looks more like this:

```
ref class StudentBase : System::Object
{
    static int number;
    StudentBase() : System::Object()
    {
    }
    static StudentBase()
    {
    }
};
```

A value type can't declare a default constructor because the CLR can't guarantee that any default constructors on value types will be called appropriately, although members are 0-initialized automatically by the CLR. In any case, a value type should be a simple type that exhibits value semantics, and it shouldn't need the complexity of a default constructor—or even a destructor, for that matter. Note that in addition to not allowing default constructors, value types can't have user-defined destructors, copy constructors, and copy-assignment operators.

Before you end up concluding that value types are useless, you need to think of value types as the POD equivalents in the .NET world. Use value types just as you'd use primitive types, such as `int`s and `char`s, and you should be OK. When you need simple types, without the complexities of virtual functions, constructors and operators, value types are the more efficient option, because they're allocated on the stack. Stack access will be faster than accessing an object from the garbage-collected CLR heap. If you're wondering why this is so, the stack implementation is far simpler when compared to the CLR heap. When you consider that the CLR heap also intrinsically supports a complex garbage-collection algorithm, it becomes obvious that the stack object is more efficient.

It must be a tad confusing when I mention how value types behave differently from reference types in certain situations. But as a developer, you should be able to distinguish the conceptual differences between value types and reference types, especially when you design complex class hierarchies. As we progress through this book and see more examples, you should feel more comfortable with these differences.

Because we've already talked about constructors, we'll discuss copy constructors next.

1.5.3 Copy constructors

A *copy constructor* is one that instantiates an object by creating a copy of another object. The C++ compiler generates a copy constructor for your native classes, even if you haven't explicitly done so. This isn't the case for managed classes. Consider the following bit of code, which attempts to copy-construct a ref object:

```
ref class Student
{
};

int main(array<System::String^>^ args)
{
    Student^ s1 = gcnew Student();
    Student^ s2 = gcnew Student(s1);    <-- (1)
}
```

If you run that through the compiler (1), you'll get compiler error C3673 (*class does not have a copy-constructor*). The reason for this error is that, unlike in standard C++, the compiler won't generate a default copy constructor for your class. At least one reason is that all ref objects implicitly derive from `System::Object`, which doesn't have a copy constructor. Even if the compiler attempted to generate a copy constructor for a ref type, it would fail, because it wouldn't be able to access the base class copy constructor (it doesn't exist).

To make that clearer, think of a native C++ class `Base` with a private copy constructor, and a derived class `Derived` (that publicly inherits from `Base`). Attempting to copy-construct a `Derived` object will fail because the base class copy constructor is inaccessible. To demonstrate, let's write a class that is derived from a base class that has a private copy constructor:

```
class Base
{
public:
    Base(){}
private:
    Base(const Base&);
};

class Derived : public Base
{
};

int _tmain(int argc, _TCHAR* argv[])
{
    Derived d1;
    Derived d2(d1); // <-- won't compile
}
```

Because the base object's copy constructor is declared as `private` and therefore is inaccessible from the derived object, this code won't compile: The compiler is unable to copy-construct the derived object. What happens with a ref class is similar to this code. In addition, unlike native C++ objects, which aren't polymorphic unless you access them via a pointer, ref objects are implicitly polymorphic (because they're always accessed via reference handles to the CLR heap). This means a compiler-generated copy constructor may not always do what you expect it to do. When you consider that ref types may contain member ref types, there is the question of whether a copy constructor implements shallow copy or deep copy for those members. The VC++ team presumably decided that there were too many equations to have the compiler automatically generate copy constructors for classes that don't define them.

If you want copy-construction support for your class, you must implement it explicitly, which fortunately isn't a difficult task. Let's add a copy constructor to the `Student` class:

```
ref class Student
{
public:
    Student(){}
    Student(const Student^)
    {
    }
};
```

That wasn't all that tough, was it? Notice how you have to explicitly add a default parameterless constructor to the class. This is because it won't be generated by the compiler when the compiler sees that there is another constructor present. One limitation with this copy constructor is that the parameter has to be a `Student^`, which is OK except that you may have a `Student` object that you want to pass to the copy constructor. If you're wondering how that's possible, C++/CLI supports stack semantics, which we'll cover in detail in chapter 3. Assume that you have a `Student` object `s1` instead of a `Student^`, and you need to use that to invoke a copy constructor:

```
Student s1;
Student^ s2 = gcnew Student(s1); //error C3073
```

As you can see, that code won't compile. There are two ways to resolve the problem. One way is to use the unary `%` operator on the `s1` object to get a handle to the `Student` object:

```
Student s1;
Student^ s2 = gcnew Student(%s1);
```

Although that compiles and solves the immediate problem, it isn't a complete solution when you consider that every caller of your code needs to do the same thing if they have a `Student` object instead of a `Student^`. An alternate solution is to have two overloads for the copy constructor, as shown in listing 1.2.

Listing 1.2 Declaring two overloads for the copy constructor

```
ref class Student
{
//...
public:
    Student(){}
    Student(String^ str):m_name(str){}
    Student(const Student^)    <-- 1
    {
    }
    Student(const Student%)    <-- 2
    {
    }
};

//...

    Student s1;
    Student^ s2 = gcnew Student(s1);
```

This solves the issue of a caller requiring the right form of the object, but it brings with it another problem: code duplication. You could wrap the common code in a `private` method and have both overloads of the copy constructor call this method, but then you couldn't take advantage of initialization lists.

Eventually, it's a design choice you have to make. ❶ If you only have the copy constructor overload taking a `Student^`, then you need to use the unary `%` operator when you have a `Student` object; and ❷ if you only have the overload taking a `Student%`, then you need to dereference a `Student^` using the `*` operator before using it in copy construction. If you have both, you may end up with possible code duplication; and the only way to avoid code duplication (using a common function called by both overloads) deprives you of the ability to use initialization lists.

My recommendation is to use the overload that takes a handle (in the previous example, the one that takes a `Student^`), because this overload is visible to other CLI languages such as C# (unlike the other overload)—which is a good thing if you ever run into language interop situations. The unary `%` operator won't really slow down your code; it's just an extra character that you need to type. I also

suggest that you stay away from using two overloads, unless it's a specific case of a library that will be exclusively used by C++ callers; even then, you must consider the issue of code duplication.

Now you know that if you need copy construction on your ref types, you must implement it yourself. So, it may not be surprising to see in the next section that the same holds true for copy-assignment operators.

1.5.4 Assignment operators

The copy-assignment operator is one that the compiler generates automatically for native classes in standard C++, but this isn't so for a ref class. The reasons are similar to those that dictate that a copy constructor isn't automatically generated. The following code (using the `Student` class defined earlier) won't compile:

```
Student s1("Nish");
Student s2;
s2 = s1; // error C2582: 'operator =' function
           // is unavailable in 'Student'
```

Defining an assignment operator is similar to what you do in standard C++, except that the types are managed:

```
Student% operator=(const Student% s)
{
    m_name = s.m_name;
    return *this;
}
```

Note that the copy-assignment operator can be used only by C++ callers, because it's invisible to other languages like C# and VB.NET. Also note that, for handle variables, you don't need to write a copy-assignment operator, because the handle value is copied over intrinsically.

You should try to bring many of the good C++ programming practices you followed into the CLI world, except where they aren't applicable. As an example, the assignment operator doesn't handle self-assignment. Although it doesn't matter in our specific example, consider the case in listing 1.3.

Listing 1.3 The self-assignment problem

```
ref class Grades
{
    //...
};

ref class Student
{
```

❶ Class with nontrivial ctor/dtor

```
    String^ m_name;
    Grades^ m_grades;
public:
    Student(){}
    Student(String^ str):m_name(str){}
    Student% operator=(const Student% s)
    {
        m_name = s.m_name;
        if(m_grades) [#2]
            delete m_grades;
        m_grades = s.m_grades;
        return *this;
    }
    void SetGrades(Grades^ grades)
    {
        //...
    }
};
```

❷ **Possible problem if self-assignment occurs**

In the preceding listing, ❶ assume that `Grades` is a class with a nontrivial constructor and destructor; thus, in the `Student` class assignment operator, before the `m_grades` member is copied, ❷ the existing `Grades` object is explicitly disposed by calling `delete` on it—all very efficient. Let's assume that a self-assignment occurs:

```
while(some_condition)
{
    // studarr is an array of Student objects
    studarr[i++] = studarr[j--]; // self-assignment occurs if i == j
    if(some_other_condition)
        break;
}
```

In the preceding code snippet, if ever `i` equals `j`, you end up with a corrupted `Student` object with an invalid `m_grades` member. Just as you would do in standard C++, you should check for self-assignment:

```
Student% operator=(const Student% s)
{
    if(%s == this)    <-- Check for self-assignment
    {
        return *this;    <-- If it is so, return immediately
    }
    m_name = s.m_name;
    if(m_grades)
        delete m_grades;
    m_grades = s.m_grades;
    return *this;
}
```

We've covered some ground in this section—and if you feel that a lot of information has been presented too quickly, don't worry. Most of the things we've discussed so far will come up again throughout this book; eventually, it will all make complete sense to you. We'll now look at boxing and unboxing, which are concepts that I feel many .NET programmers don't properly understand—with not-so-good consequences.

1.6 Boxing and unboxing

Boxing is the conversion of a value of type `V` to an object of type `V^` on the CLR heap, which is a bit-wise copy of the original value object. Figure 1.4 shows a diagrammatic representation of the boxing process. *Unboxing* is the reverse process, where an `Object^` or a `V^` is cast back to the original value type `V`. Boxing is an implicit process (although it can be explicitly forced, as well), whereas unboxing is always an explicit process. If it sounds confusing, visualize a real box into which you put some object (say, a camera) so that you can send it via FedEx to your friend the next city. The same thing happens in CLR boxing. When your friend receives the package, they open the box and retrieve the camera, which is analogous to CLR unboxing.

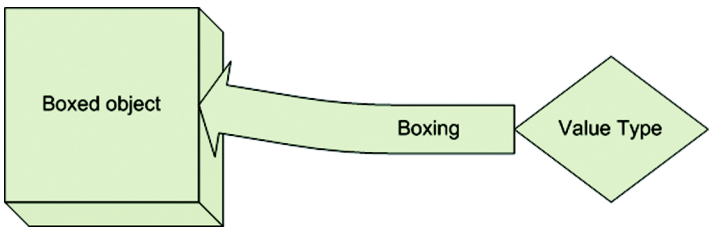

Figure 1.4
The boxing process

In this section, we'll look at how boxing is an implicit operation in the new C++/CLI syntax, how boxing ensures type safety, how boxing is implemented at the MSIL level, and how to assign a `nullptr` to a boxed value type.

1.6.1 Implicit boxing in the new syntax

Whenever you pass a simple type like an `int` or a `char` to a method that expects an `Object`, the `int` or `char` is boxed to the CLR heap, and this boxed copy is used by the method. The reason is that ref types are always references to whole objects on the CLR heap, whereas value types are typically on the stack or even on the native C++ heap. When a method expects an `Object` reference, the value type has to be

copied to the CLR heap, where it must behave like a regular ref-type object. In the same way, when the underlying value type has to be retrieved, it must be unboxed back to the original value-type object. The internal boxing and unboxing mechanisms are implemented by the CLR and supported in MSIL, so all the compiler needs to do is emit the corresponding MSIL instructions.

In the old syntax, boxing was an explicit process using the `__box` keyword. Several programmers complained about the extra typing required. Because most people felt that the double-underscored keywords were repulsive, the fact that they had to use one of those keywords a gratuitous number of times in the course of everyday programming made them all the more upset. You can't blame them, as the code examples in table 1.5 show.

Table 1.5 Boxing differences between the old and new syntaxes

Explicit boxing in the old MC++ syntax	Implicit boxing in C++/CLI

`int __box* AddNums(`
`  int __box* i, int __box* j)`
`{`
`  return __box(*i + *j);`
`}`
 	
`//...`
 	
`int i = 17, j = 23;`
`int sum = *AddNums(`
`  __box(i), __box(j));`
`Console::WriteLine(`
`  "The sum is {0}", __box(sum));`
	`int^ AddNums(int^ i, int^ j)`
	`{`
	`  return *i + *j;`
	`}`
	
	`//...`
	
	`int i = 17, j = 23;`
	`int sum = *AddNums(i, j);`
	`Console::WriteLine(`
	`  "The sum is {0}", sum);`

It would be an understatement to say that the second code example is a lot more pleasing to the eye and involves much less typing. But implicit boxing has a dangerous disadvantage: It hides the boxing costs involved from the programmer, which can be a bad thing. Boxing is an expensive operation; a new object has to be created on the CLR heap, and the value type must be bitwise copied into this object. Similarly, whenever a lot of boxing is involved, chances are good that quite a bit of unboxing is also being performed. Unboxing typically involves creating the original value type on the managed stack and bitwise copying its data from the boxed object. As a developer, if you ignore the costs of repeated boxing/unboxing operations, either knowingly or unknowingly, you may run into performance issues, most often in applications where performance is a major concern.

1.6.2 Boxing and type-safety

When you box a value type, the boxed copy is a separate entity from the original value type. Changes in one of them won't be reflected in the other. Consider the following code snippet, where you have an `int`, a boxed object containing the `int`, and a second `int` that has been explicitly unboxed from the boxed object:

```
int i = 100;
Object^ boxed_i = i; //implicitly boxed to Object^
int j = *safe_cast<int^>(boxed_i); //explicitly unboxed
Console::WriteLine("i={0}, boxed_i={1}, j={2}", i, boxed_i, j);
i++; j--;
Console::WriteLine("i={0}, boxed_i={1}, j={2}", i, boxed_i, j);
```

The first call to `Console::WriteLine` outputs

```
i=100, boxed_i=100, j=100
```

The second call outputs

```
i=101, boxed_i=100, j=99
```

As the output clearly indicates, they're three different entities: the original value type, the boxed type, and the unboxed value type. Notice how you had to `safe_cast` the `Object^` to an `int^` before dereferencing it. This is because dereferencing is always done on the boxed value type. To get an `int`, you have to apply the dereference operator on an `int^`, and hence the cast.

NOTE The `safe_cast` operator is new to C++/CLI and replaces `__try_cast` in the old syntax. `safe_cast` is guaranteed to produce verifiable MSIL. You can use `safe_cast` wherever you would typically use `dynamic_cast`, `reinterpret_cast`, or `static_cast`. At runtime, `safe_cast` checks to see if the cast is valid; if so, it does the conversion or else throws a `System::InvalidCastException`.

When you box a value type, the boxed value remembers the original value type—which means that if you attempt to unbox to a different type, you'll get an `InvalidCastException`. This ensures type-safety when you perform boxing and unboxing operations. Consider listing 1.4, which demonstrates what happens when you attempt to unbox objects to the wrong value types.

Listing 1.4 Type-safety in boxing

```
int i = 100;
double d = 55.673;

Object^ boxed_int = i; //box int to Object^
Object^ boxed_double = d; //box double to Object^

try
{
    int x = *safe_cast<int^>(boxed_double); //compiles fine
}
catch(InvalidCastException^ e) //exception thrown at runtime
{
    Console::WriteLine(e->Message);
}

try
{
    double x = *safe_cast<double^>(boxed_int); //compiles fine
}
catch(InvalidCastException^ e) //exception thrown at runtime
{
    Console::WriteLine(e->Message);
}
```

This listing attempts to unbox a boxed `double` to an `int` and a boxed `int` to a `double`. Although the code compiles, during runtime, an `InvalidCastException` is thrown:

```
Unable to cast object of type 'System.Double' to type 'System.Int32'.
Unable to cast object of type 'System.Int32' to type 'System.Double'.
```

Note that type matching is always exact; for instance, you can't unbox an `int` into an `__int64` (even though it would be a safe conversion).

1.6.3 Implementation at the MSIL level

MSIL uses the `box` instruction to perform boxing. The following is a quote from the MSIL documentation: "The box instruction converts the raw valueType (an unboxed value type) into an instance of type Object (of type O). This is accomplished by creating a new object and copying the data from valueType into the newly allocated object."

To get a better idea of how boxing is done, let's look at how the MSIL is generated (see table 1.6).

Table 1.6 MSIL generated for a boxing operation

C++/CLI code	Generated MSIL

	`.locals init (`
	`  [0] int32 i,`
	`  [1] object o)`

`int i = 100;`	`IL_0000: ldc.i4.s 100` `IL_0002: stloc.0`

`Object^ o = i;`	`IL_0003: ldloc.0`
	`IL_0004: box int32`
	`IL_0009: stloc.1`

We won't decipher each MSIL instruction, but the line of code that is of interest is the instruction at location IL_0004: `box int32`. The instruction before it, `ldloc.0`, loads the contents of the local variable at the 0th position (which happens to be the `int` variable `i`) into the stack. The `box` instruction creates a new `Object`, copies the value (from the stack) into this object (using bitwise copy semantics), and pushes a handle to this `Object` on the stack. The `stloc.1` instruction pops this `Object` from the stack into the local variable at the first position (the `Object^` variable `o`). Table 1.7 shows how unboxing is done at the MSIL level.

Unboxing is the reverse process. The `Object` to be unboxed is pushed on the stack, and a cast to `int^` is performed using the `castclass` instruction. On successful completion of this call, the `Object` on the stack is of type `int^`. The `unbox int32` instruction is executed, and it converts the boxed object (on the stack) to a managed pointer to the underlying value type. This behavior is different from

Table 1.7 MSIL generated for unboxing

C++/CLI code	Generated MSIL

	`.locals init (`
	`  [0] int32 x,`
	`  [1] object o)`

`int x = *safe_cast<int^>(o);`	`IL_0000: ldloc.1`
	`IL_0001: castclass int32`
	`IL_0006: unbox int32`
	`IL_000b: ldind.i4`
	`IL_000c: stloc.0`

boxing where a new object is created; however, `unbox` doesn't create a new value type instance. Instead, it returns the address of the underlying value type on the CLR heap. The `ldind.i4` instruction indirectly loads the value from the address returned on the stack by the `unbox` instruction. This is basically a form of dereferencing. Finally, the `stloc.0` instruction stores this value in local variable 0, which happens to be the `int` variable `x`.

The basic purpose of showing you the generated IL is to give you a better idea of the costs involved in boxing/unboxing operations. When you box, you incur the cost of creating a new `Object`. When copying the value type into this `Object`, you waste CPU cycles as well as extra memory. When you unbox, you typically have to `safe_cast` to your value type's corresponding handle type, and the runtime has to check to see if it's a valid cast operation. Once you do that, the actual unboxing reveals the address of the value type object within the CLI object, which has to be dereferenced and the original value copied back.

Thus, both boxing and unboxing are very expensive operations. You probably won't see much of a performance decrease for simple applications, but bigger and more complex applications may be seriously affected by performance loss if you don't restrict the number of boxing/unboxing operations that are performed. Because boxing is implicit now, as a programmer you have to be that little bit extra-cautious when you convert value types to ref types, either directly or indirectly, as when you call a method that expects a ref-type argument with a value type.

1.6.4 Assigning null to a boxed value type

An interesting effect of implicit boxing is that you can't initialize a boxed value type to null by assigning a 0 to it. You have to use the `nullptr` constant to accomplish that:

```
int^ x1 = nullptr;
if(!x1)
    Console::WriteLine("x1 is null"); // <-- this line is executed
else
    Console::WriteLine("x1 is not null");

int^ x2 = 0;
if(!x2)
    Console::WriteLine("x2 is null");
else
    Console::WriteLine("x2 is not null");
      // ^-- this line is executed
```

In the second case, the 0 is treated as an `int`, which is boxed to an `int^`. You specifically need to use `nullptr` if you want to assign a handle to null. Note that the compiler issues warning C4965 (*implicit box of integer 0; use nullptr or explicit cast*) whenever it can.

When you have two overloads for a function that differ only by a value type argument, with one overload using the value type and the other using the boxed value type, the overload using the value type is given preference:

```
void Show(int)
{
    Console::WriteLine(__FUNCSIG__);
}
void Show(int^)
{
    Console::WriteLine(__FUNCSIG__);
}
```

Now, a call such as

```
Show(75);
```

will call the `Show(int)` overload instead of the `Show(int^)` overload. Keep in mind that, when selecting the best overload, the compiler gives lowest priority to one that requires boxing. If you had another overload that required a nonboxing cast, that overload would receive preference over the one that requires boxing. Given three overloads, one that takes an `int`, one that takes a `double`, and one that takes an `int^`, the order of precedence would be

```
[first] void Show(int)
[second] void Show(double)
[third] void Show(int^)
```

To force an overload, you can do a cast to the argument type for that overload:

```
Show(static_cast<int^>(75));
```

Now that we've covered boxing and unboxing, I suggest that you always consciously keep track of the amount of boxing that is done in your code. Because it's an implicit operation, you may miss out on intensive boxing operations; but where boxing occurs, there is bound to be unboxing, too. If you find yourself having to do a lot of unboxing, review your code to see if it overuses boxing, and try to redesign your class to reduce the amount required. Take special care inside loops, which is where most programmers end up with boxing-related performance issues.

1.7 Summary

In this chapter, we've covered some of the fundamental syntactic concepts of the C++/CLI language. As you may have inferred by now, the basic programming concepts remain the same in C++/CLI as in standard C++. However, you need to accommodate for the CLI and everything that comes with it, such as garbage collection, handles to the CLR heap, tracking references, and the implicit derivation from `System::Object`. Topics we've covered included how to declare and instantiate CLI types; how to use handles, and how they differ from pointers; and how boxing and unboxing are performed when converting from value types to reference types.

The designers of C++/CLI have gone to great lengths to ensure as close a similarity to the standard C++ language as is practically possible, but some changes had to be made due to the nature of the CLI (which is a different environment from native C++). As long as you're aware of those differences and write code accordingly, you'll do well.

Although C++/CLI's biggest strength is its ability to compile mixed-mode code, you need to be familiar with the core CLI programming concepts before you can begin writing mixed-mode applications. With that view, in the next couple of chapters we'll explore the CLI features that are supported by C++/CLI, such as properties, delegates and events, CLI arrays, CLI pointers, stack semantics, function overriding, generics, and managed templates.

.NET

C++/CLI IN ACTION

Nishant Sivakumar

Developers initially welcomed Microsoft's Managed C++ for .NET, but the twisted syntax made it difficult to use. Its much-improved replacement, C++/CLI, now provides an effective bridge between the native and managed programming worlds. Using this technology, developers can combine existing C++ programs and .NET applications with little or no refactoring. Accessing .NET libraries like Windows Forms, WPF, and WCF from standard C++ is equally easy.

C++/CLI in Action is a practical guide that will help you breathe new life into your legacy C++ programs. The book begins with a concise C++/CLI tutorial. It then quickly moves to the key themes of native/managed code interop and mixed-mode programming. You'll learn to take advantage of GUI frameworks like Windows Forms and WPF while keeping your native C++ business logic. The book also covers methods for accessing C# or VB.NET components and libraries. Written for readers with a working knowledge of C++.

What's Inside

- Call C++ libraries from C# or VB.NET
- C++ for WPF and WCF
- Mixed-mode programming techniques
- Move from Managed C++ to C++/CLI

Nishant Sivakumar is a C++ developer living in Atlanta, GA. He has been a Microsoft Visual C++ MVP since 2002 and is active online at blog.voidnish.com.

For more information, code samples, and ebook visit
www.manning.com/Cplusplus/CLIinAction

"… a great resource, an outstanding job, a must-read …"

—Ayman B. Shoukry
VC++ Team
Microsoft Corporation

"… explains the C++/CLI language and shows how it can be used."

—James Johnson
www.misterdotnet.com/blog/

"… a great reference … a must-have in any C++ programmer's bookshelf."

—Michael L. Taylor, C# MVP

"… absolutely the only book you'll need to learn this powerful new .NET language … a lifesaver …."

—Christian Graus, Code Project

ISBN-10: 1-932394-81-8
ISBN-13: 978-1-932394-81-8

$49.99 US/$64.99 Canada