

Unedited
Draft



C++/CLI IN ACTION

Nishant Sivakumar

 MANNING



C++/CLI in Action
by Nishant Sivakumar

Unedited Draft Chapter 1

Copyright 2006 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Introduction to C++/CLI

When C++ was wedded to CLI with a slash, it was pretty obvious pretty fast that it wasn't going to be one of those celebrity marriages. Put simply, the world's most powerful high level programming language - C++ - was given a face-lift so that it could be used to develop for what could potentially be the world's most popular run-time environment – the CLI.

In this chapter, we'll see what C++/CLI can be used for, and how C++/CLI improves over the now obsolete Managed C++ syntax. We'll also go over some basic C++/CLI syntax, and by the end of this chapter, you will know how to write and compile a C++/CLI program, and how to declare and use managed types. Some of the new syntactic features might take a little getting used to, but then 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

You probably already know about both C++ and CLI, but let's briefly visit them before discussing their merger. C++ is a versatile programming language with a substantial number of features that make it the most powerful and flexible coding tool for a professional developer. The CLI (Common Language Infrastructure) is an architecture that supports a dynamic language-independent programming model based on a Virtual Execution System, and 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 Common Language Infrastructure. Figure 1.1 shows the relation between standard C++ and the Common Language Infrastructure.

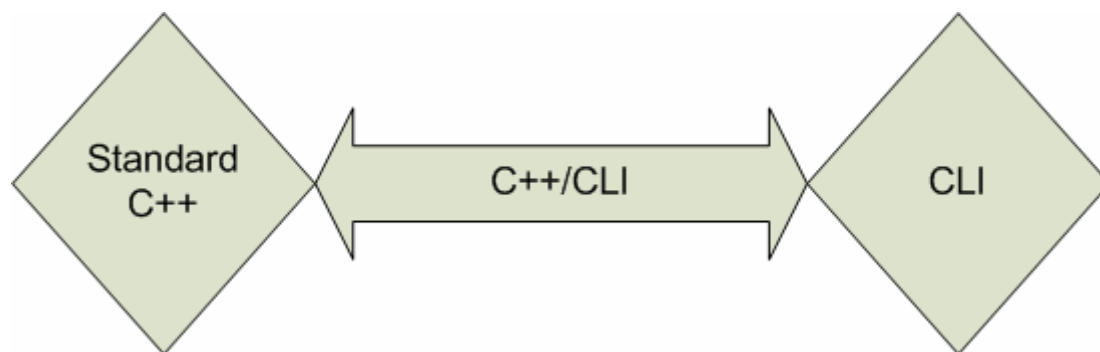


Figure 1.1 How C++/CLI connects standard C++ to the CLI

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:

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

2. C++/CLI allows the reuse of a developer's native C++ code base, saving the agony of having to rewrite all his existing code to enable it to run on the .NET Framework.
3. 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*".

Visual C++ 2005 is not 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++), but 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 look at why it is such an invaluable inclusion among CLI languages.

The .NET Framework

It's important for you as a reader, to have a basic understanding of the .NET Framework, because while this book will teach you the C++/CLI syntax before we move on to various interop mechanisms and strategies, it does not attempt to teach you the core details of the .NET Framework. If you have never used the .NET Framework previously, you should take a look at the book's *Appendix A : A Concise Introduction to the .NET Framework* before you proceed further with this book. On the other hand, if you have worked with the .NET Framework a while ago, take a look at these quick definitions (in no particular order) of various terms associated with the .NET Framework that you will encounter in this and other chapters, just to refresh your memory.

- **.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 CLR (Common Language Runtime) and the BCL (Base Class Library).
- **The CLR:** The CLR is the core of the .NET Framework and implements the fundamental aspects of the CLI such as the virtual execution system, the garbage collector, and the Just In Time (JIT) compiler.
- **The BCL:** The Base Class Library is an extensive set of .NET classes that are used by any .NET language (such as C#, VB.NET, C++/CLI etc.)
- **VES:** The Virtual Execution System is the engine that's responsible for executing managed code, including 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 which will free resources when they are no longer needed, freeing the developer from the need to do so.
- **JIT Compiler:** .NET compilers (C#, VB.NET, C++/CLI etc.) compile source code into an intermediate language called MSIL (Microsoft Intermediate Language) and the CLR, at runtime, uses the JIT compiler to compile this MSIL into the native code for the underlying Operating System, before executing it.
- **CTS:** The CTS (Common Type System) is a set of rules that specify how the CLR can define, use, create and manage types.
- **CLS:** The CLS (Common Language Specification) is a subset of the CTS that all languages must implement if they are to be considered CLS-compliant. CLS-compliant languages can interop with each other as long as they do not use any non-CLS-compliant features present in their specific compiler version.

Again, I'd like to reiterate that if you are still not feeling very familiar with these terms, or if you wish to understand them in a little more detail, please take a detour into *Appendix A* where most of these things are explained a little more elaborately.

1.1.1 What C++/CLI can do for you

If you are reading this book, chances are good that you are looking to move your applications to the .NET Framework. The biggest concern C++ developers have when they are looking to make the move to .NET is that they're afraid of abandoning their existing native code 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 would have to change a small percentage of your code to successfully compile and build your applications. But this is definitely a far better option than either abandoning all your code or rewriting it entirely. Once you have successfully compiled your code for the CLR, your 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 instance, you can take a regular MFC Dialog Based application and give it some encryption functionality using the .NET cryptography classes. You are not restricted to managed libraries, because C++/CLI lets you seamlessly interop between managed and native code.

Most powerful language for interop

While other languages like C# and VB.NET do 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, so very often, you 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 have done. Compare that with C# or VB.NET where you are 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 very popular use of interop is to access new managed frameworks like Windows Forms from existing native applications (Note that this 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, and you have recently acquired an outstanding Windows Forms based UI library from another company. Take VC++ 2005, recompile the MFC application for the CLR, change the UI layer to use the Windows Forms library, and now you have the same application that uses the same underlying

business logic, but with the new shiny user interface. You are not restricted to just Windows Forms, or even to UI Frameworks for that matter. For instance, the next version of Windows (called Windows Vista) will introduce an entirely new UI framework called the *Windows Presentation Foundation* (WPF). It's a managed framework and C++/CLI will allow you to access that from existing native applications. So, when Vista is released, your applications will be able to flaunt the Windows Vista look and feel. Another powerful managed framework that is coming out in Vista is called the *Windows Communication Foundation* (WCF), which as the name implies is a powerful communication framework written in managed code. And yes, though I guess you knew I was going to say that, you can access the WCF from your Visual C++ applications. While 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, and it is to MSIL what C used to be to Assembly Language in the old days. C++/CLI is also the only CLI language currently, 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 they just continue to use the old MC++ syntax. That's what we'll look into next.

1.1.2 The rationale behind the new syntax

The Managed Extensions to C++ introduced in VC++ 2002 were not well-accepted by the C++ developer community. While most people appreciated the fact that they could use C++ for .NET development, pretty much everybody felt that the syntax was gratuitously twisted and unnatural, that the managed and unmanaged pointer usage semantics was pretty confusing, and that C++ hadn't been given equal footing as a CLI language with other languages like C# or VB.NET.

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 Common Language Infrastructure (CLI). Microsoft developed and submitted full draft specifications of a binding of the "C++ Programming Language" to the "Common Language Infrastructure" in November 2003, and C++/CLI became an international ECMA standard in December 2005; it is 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.

About the ECMA

The ECMA (which originally expanded to 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 is not used anymore and the body today goes by the name ECMA International. You can visit their 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 look at how C++/CLI improves on these issues. If you have used the old syntax in the past, you will definitely

appreciate the enhancements in the new syntax, and if you haven't, you'll still notice the stark difference in syntactic 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 were not going to feel very comfortable using that syntax, and most C++ developers simply chose not to use a syntax that they felt awkward about.

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

Having used both forms of the language, I must say that with the new syntax, I don't feel as much like a fish out of water as I did with the old one. Take a look at the following table where I have compared the old and new syntaxes, and you will see what I mean.

Table 1.1 Comparison between old and new syntaxes

Old syntax	C++/CLI syntax
<pre>__gc __interface I { };</pre>	<pre>interface class I { };</pre>
<pre>__delegate int ClickHandler();</pre>	<pre>delegate int ClickHandler();</pre>
<pre>__gc class M : public I { __event ClickHandler* OnClick; public: __property int get_Num() { return 0; } __property void set_Num(int) { } };</pre>	<pre>ref class M : public I { event ClickHandler^ OnClick; public: property int Num { int get() { return 0; } void set(int value) { } } };</pre>

Even without knowing the syntactic rules for either the old syntax or C++/CLI, you shouldn't find it very hard to decide which of the above syntaxes 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 in this chapter we will go through the fundamental syntactic concepts of the C++/CLI language. I just wanted to show you why the old syntax never got popular and how Microsoft has improved on the look-and-feel aspects of the syntax in the new C++/CLI specification.

With the old syntax, every time you use a CLI feature, such as a delegate or a property, you have to prefix it with those underscored keywords. For a property definition, the old syntax required separate setter and getter blocks, and does not syntactically organize them into a single block, which means that if

you carelessly separated the getter and setter methods with some other code, there's no visual cue that they are part of the same property definition; whereas with the new syntax, you just put your getter and setter functions inside a property block, so the relationship between them is visually maintained. To summarize my personal thoughts on this, with the old syntax, you feel that you are using two unrelated sub-languages (one for managed and one for native code) with a single compiler, whereas with the new syntax, you feel that you are using C++, albeit with a lot of new keywords, but it's still one single language.

Programmers can be a bunch of compiler-snobs, and many developers opined that C# and VB.NET were proper .NET languages, while 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 like 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, which was not such 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 he has to repeatedly call `MoveNext` till it returns `false`, and this is information that should have been hidden from him or her. Requiring the internal implementation details of the collection to be directly used beats the purpose of having collection classes, when the very purpose of having them is to abstract the internal details of an enumerable collection class from the programmer.

Now look at equivalent C++/CLI code :

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

By adding constructs like `for each`, which allow developers a more natural syntax to access .NET features, what the VC++ team has done is to give us, the developers, a cozy feeling that C++/CLI is now a first class language for .NET programming. Later on in this chapter, we will see that boxing is now implicit, which means you don't have to use the gratuitous `__box` keyword that you had to in the old syntax, and if you don't know what boxing means, don't worry, because that 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 like templates and deterministic destruction were not available, and most C++ developers felt severely handicapped by the apparent feature reductions when using MC++.

With C++/CLI, templates are now supported on both managed and unmanaged types, and in addition, C++/CLI is the only CLI language that supports stack semantics and deterministic destruction (though 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 like 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 totally 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 totally different things. The new call on the native class N results in the C++ new operator being called while the new call on the managed class R is compiled into the MSIL newobj instruction. While the native object is allocated on the C++ heap, 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. So while the R* object looks like a native C++ pointer, it does not behave like one and its address cannot be assumed to remain fixed. The good news is that this has been fixed in C++/CLI; we now have an additional gcnew keyword for instantiating managed objects and 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 on in this chapter, we will take a more detailed look at handles and the gcnew operator; but for now it should suffice to note that in C++/CLI, there will be not be any managed/unmanaged pointer confusion.

Unverifiable code

The Managed C++ compiler could not produce verifiable code, which meant that you could not use it to write any code that was to run under a protected environment – for example, as a SQL Server stored procedure. Visual C++ 2005 supports a special compiler mode (/clr:safe) which produces verifiable code,

and actually disallows you from compiling any non-verifiable code by generating errors during compilation. The advantage to being able to create verifiable assemblies is that the CLR can enforce active CLR security restrictions on the running application and 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 even be permitted to execute.

It should be pretty obvious by now why Microsoft decided to bring out a new syntax, and if you've never used the old syntax, you can consider yourself lucky. Now you can straight away use the new and powerful 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 (which is available in VC++ 2005 through the `/clr:oldSyntax` compiler switch) is not guaranteed to be available in future VC++ versions nor will there be any significant improvements done on it. Let's now move onto our first C++/CLI program.

1.2 Hello World in C++/CLI

Whenever I look at a new language or compiler, I'm always eager to get my first program compiling fine and so before we go any further, let's write our first Hello World application in C++/CLI. In this section, we will also take a look at the new compiler options that have been introduced in VC++ 2005 to support compilation for managed code. There's nothing overly complicated about the code in Listing 1.1, but for our purposes, it will do 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]; #1
    DWORD size = UNLEN + 1; #1
    GetUserName(buffer, &size); #1
    String^ greeting = "Hello"; #2
    Console::WriteLine("{0} {1}", #2
        greeting, gcnew String(buffer));
    return 0;
}
```

(#1) : <Get the current user>

(#2) : <Display a greeting>

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

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

That's all. You can run it now and it will promptly display "Hello" followed by the current user (most likely your Windows login name) on the console. Except for the `gcnew` keyword (which we will talk about a while 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 will be JIT (just in time) compiled and executed by the CLR just like any executable you might create using C#, VB.NET or any other CLI language. While this is a rather small example, it still communicates two very important facts, that 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 that you can write managed [#2] and native code [#1] within the same application.

For those of you not familiar with the .NET Framework, `Console` is a .NET Framework BCL (Base Class Library) class which belongs to the `System` namespace (hence the `using namespace` declaration on top), and `WriteLine` is a static method of the `Console` class. In the preceding listing, we have used native data types like `TCHAR` and `DWORD` as well as managed data types like `String`, and similarly, we have used a native Win32 API call (`GetUserName`) as well as a managed class (`System::Console`), and the best part is that we have done all this in a single application (actually within a single function in this case). Though you may not have realized it yet, you've just written a mixed-mode application that mixes native and managed code. Congratulations! There's a lot that you can do with mixed-mode coding, and we'll see far more useful applications of that for most of the later portions of this book.

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

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 like managed types and garbage collection. You can specify sub-options to the `/clr` option to further specify the type of assembly that you want created. The following table is a partial list of the `/clr` sub-options that you can specify and what they do; I have only included those sub-options that are of interest to us. For a more complete list, you should refer MSDN documentation for the C++ compiler command line switches.

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

Compiler switch	Description
<code>/clr</code>	Creates an assembly targeting the Common Language Runtime. The output file may contain both MSIL as well as native code (mixed mode assemblies). This is the most commonly used switch (which is probably why it's the default) and allows us to 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 our most commonly used compilation mode throughout this book.
<code>/clr:pure</code>	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. For C# developers, you can think of this as being equivalent to using the C# compiler in unsafe mode – the output is pure MSIL but not necessarily verifiable.
<code>/clr:safe</code>	Creates an MSIL-only verifiable assembly. You cannot 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.
<code>/clr:oldSyntax</code>	Enables the Managed C++ syntax available in VC++ 2002 and VC++ 2003. I would 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.

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

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

For any non-trivial program, it makes sense to use the Visual C++ development environment, though it's still good to know the various 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 really powerful development environment, and there's no reason why we shouldn't take advantage of it. For the rest of this chapter and the next two chapters, we will be using CLR enabled console applications as we take a look at the C++/CLI syntax and grammar. Those of you who want to follow along in your own console project can just type in the code as it is written in the book. (*Note that for 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 pretty straightforward and I list the steps below :

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.

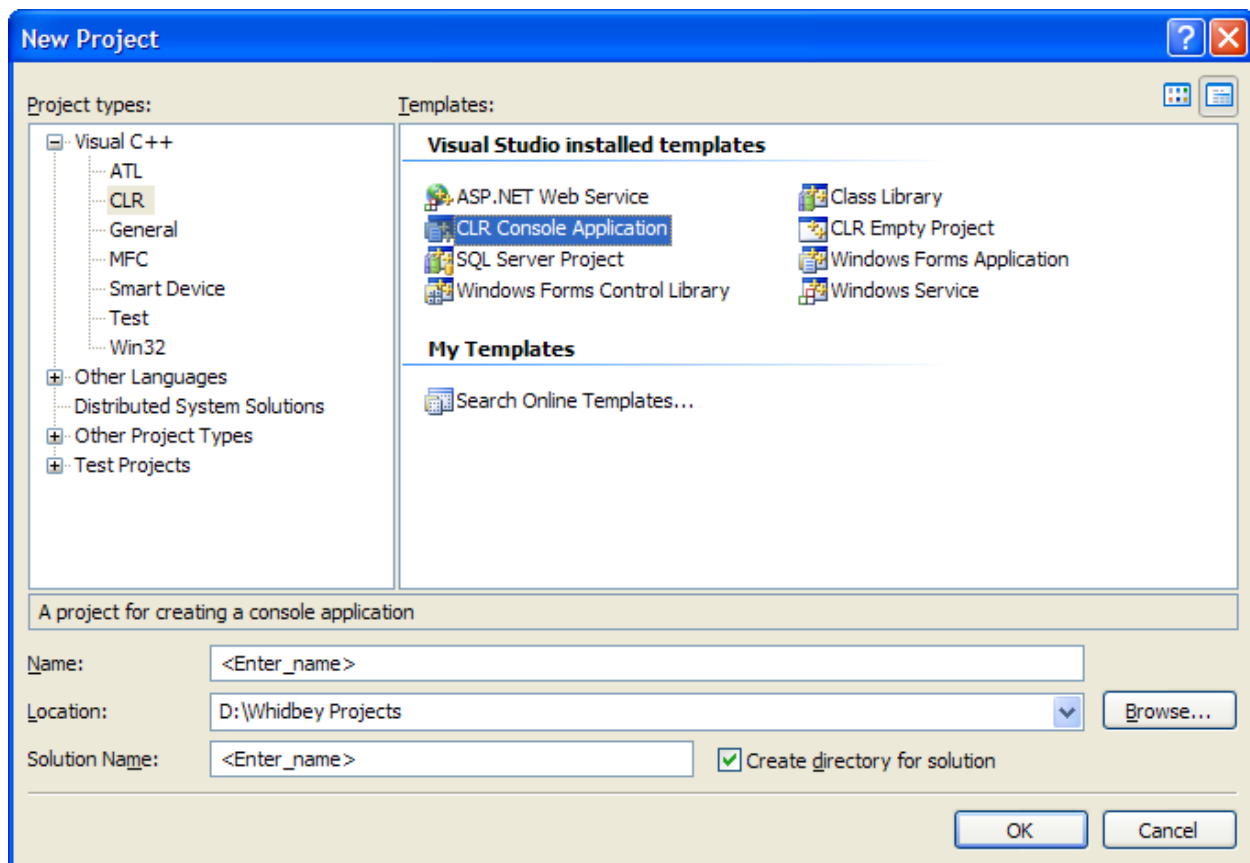


Figure 1.2 The Visual C++ 2005 New Project wizard

2. Enter a name for the project and click OK.

The wizard generates quite a few files for us, though the one that should interest us the most is the `cpp` file that has the same name as the project; so if you had named your project `Chapter01Demo`, you will see a `Chapter01Demo.cpp` file in your solution that will contain the wizard generated `main` method. You

must have used similar wizards in the past when working on MFC, ATL or even Win32 API projects, so this should be pretty straightforward to you.

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 is one that adheres to the CLI definition of a managed entry point function. The syntax might seem a little confusing right now (since we haven't begun exploring the C++/CLI syntax), but args is essentially a managed array of System::String objects that represent the command line arguments passed to the application. An important distinction that should be kept in mind is that, unlike the native C++ main prototypes, the name of the program is not passed as the 0-indexed argument. So if you run the application without any command line arguments, the array will be empty. By default, the wizard will set the project to use the /clr compilation option, but you can change that using the Project properties dialog which can be brought up from menu Project->Properties or by using the Alt-F7 keyboard shortcut (*note that the keyboard shortcuts will vary depending on your VS profile – I've used those shortcuts that are associated with the default VC++ profile*). Select General from Configuration Properties on the left, and you will 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.

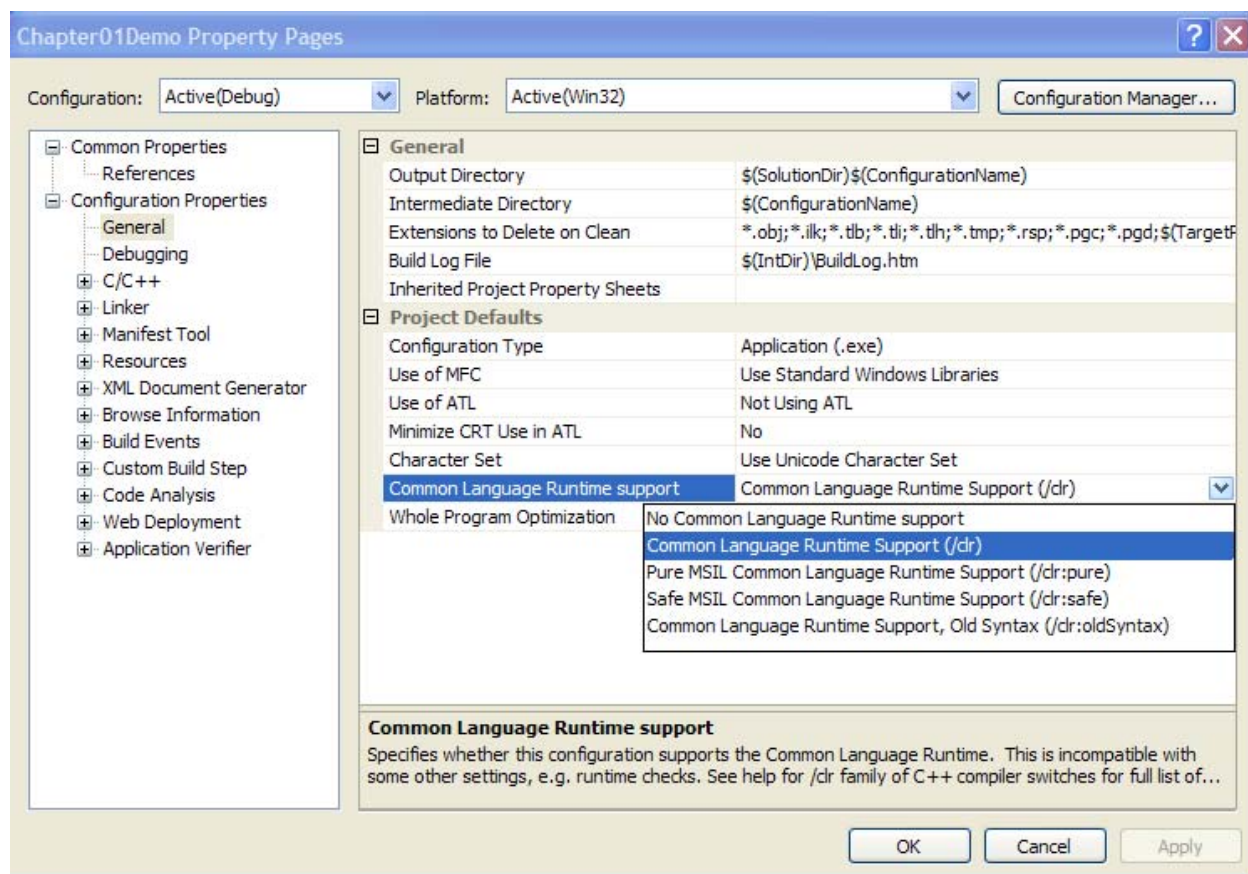


Figure 1.3 Setting the CLR compilation options using the Project Properties window

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

1.3 Declaring CLR types

In this section, we will take a 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 that is being declared.

The following table shows examples of CLI type declarations for various types.

Table 1.3 Type declaration syntax for CLI types

CLI type	Declaration syntax
Reference types	<pre>ref class RefClass1 { void Func() {} }; ref struct RefClass2 { void Func() {} };</pre>
Value types	<pre>value class ValClass1 { void Func() {} }; value struct ValClass2 { void Func() {} };</pre>
Interface types	<pre>interface class IType1 { void Func(); }; interface struct IType2 { void Func(); };</pre>

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 follow standard C++ visibility rules for structs and classes. In a class, methods are private by default and in a struct, methods are public by default; in the above table, `RefClass1::Func` and `ValClass1::Func` are both private, while `RefClass2::Func` and `ValClass2::Func` are both public. For the sake of clarity and maintaining consistence with C#, you might 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 – so that there’s no confusion at all.

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

- 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`.

[Notes] Spaced keywords

An interesting thing that you need to be aware of is that there are only 3 new reserved keywords that have been introduced in C++/CLI – `gcnew`, `nullptr` and `generic`; and all these other seemingly new keywords are actually spaced (or contextual) keywords. So, 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 here is that, any existing code that uses any of these new keywords (like `ref` or `each`) will continue 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 either `gcnew`, `nullptr` or `generic` as an identifier, C++/CLI will not compile it and you will have to rename those identifiers.

We’ve seen how CLI types can be declared. Next, we’ll see how type modifiers can be applied to these classes (or structs as the case might be).

1.3.1 Class Modifiers

You can specify the `abstract` and `sealed` modifiers on classes; and a class can be marked as both `abstract` and `sealed`, but such classes cannot be derived explicitly from any base class and can only contain `static` members. Since global functions are not 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 are wondering when and why we would need to use these modifiers, you need to remember that, to effectively write code targeting the .NET Framework, you should be able to implement every supported CLI paradigm. And 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 is not 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 as `abstract` though it does not 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 is not marked as `abstract`, such as the class shown below, 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;
};
```

But, in the generated IL, the class R1 is marked as `abstract`, which means that if you try to instantiate the class, you will get a compiler error (and you should). Not marking a class as `abstract` when it has abstract methods is rather untidy, and I strongly encourage you to explicitly mark classes as `abstract` if at least one of their methods is abstract. Note how I've used the `abstract` modifier on a class method in the above 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 cannot 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. Since a derived class can be used anywhere the base class can be used, if you allowed 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. As an example, consider a Banking application that has a `CreditCardInfo` class that is used to fetch information about an account holder's credit card transactions. Since 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 and thus any instance of the derived class is now inherently insecure. By marking the `CreditCardInfo` class as `sealed`, such a contingency can be easily avoided.

A performance benefit with using a `sealed` class is that, since the compiler knows that a `sealed` class cannot have any derived classes, it can statically resolve `virtual` member invocations on a `sealed` class instance using `non-virtual` 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 does not have to figure out which function to call, as it does

not have to determine the polymorphic type of the class (since a `CreditCardInfo` object can only be a `CreditCardInfo` object, it cannot be an object of a derived type, as there are no derived types), it directly calls the `GetHashCode` method defined by the `CreditCardInfo` class.

Here's an 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 cannot have instance methods and attempting to do so would throw compiler error C4693, which is not very puzzling when you consider that an instance method on an `abstract sealed` class would be quite worthless, as you can never have an instance of such a class. An `abstract sealed` class cannot be explicitly derived from a base class, though it implicitly derives from `System::Object`. For those of you who've used C#, it might be interesting to know that an `abstract sealed` class is the same as a C# static class.

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

1.3.2 CLI types and inheritance

Inheritance rules are pretty much similar to that in standard C++, but there are differences, and it's very important to realize what they are when using C++/CLI. The good thing is that most of the differences are obvious and natural ones that are dictated by the nature of the CLI, so you won't find it particularly strenuous to remember what they are.

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 will 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 cannot inherit from another value or ref type. That's because value types are implicitly derived from `System::ValueType`. Because CLI types do not support multiple base classes, value types cannot have any other base class. In addition, value types are always sealed, so they cannot be used as base classes. In the following code snippet, only the `Derived3` class will compile, because the other two classes attempt to inherit from a ref class and a value class, both of which are not 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 {};
```

If you are wondering why these restrictions are placed on value types, it's because value types are intended to be simple types without the complexities of inheritance or referential identity, that 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 are treated like simple POD (plain old data) 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 if 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, do not make it a value type, but if you want it to behave just as an `int` would or a `char` would, chances are good that your best option would be to declare it as a value type. Typically, you'd 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 like an `int` or `char`, you may want to make that a value type.

One important point to be aware of is that CLI types do not support multiple inheritance. So, while a CLI type can implement any number of interfaces, it can only have one immediate parent type, and if none's specified, this will implicitly be 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 and replace the `__gc` pointer concept used in Managed C++. If you remember, earlier in the chapter we had discussed the pointer usage confusion that prevailed in the old syntax. Handles solve that confusion. In my opinion, the concept of handles is one that 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, and will 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 as hat). By the way, note that when I say “punctuator” in this chapter, I am talking from a compiler perspective, and as far as the language syntax is concerned, you could 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 will use handles with managed objects allocated on the CLI 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 is not guaranteed to remain constant, and the `String` object might be moved around after a Garbage Collection cycle, but `str` will continue to be a reference to the same `System::String` object (unless 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 might look deceitfully similar to pointers but are totally different entities in 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 <code>^</code> punctuator	Pointers are denoted by the <code>*</code> punctuator
Handles are references to managed objects on the CLI heap	Pointers just point to memory addresses
Handles may refer to different memory locations depending on GC cycles and Heap Compactions.	Pointers are stable and Garbage Collection cycles do not 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 will not be updated.
Handles are type-safe.	Pointers were not designed for type-safety.
The <code>gcnew</code> operator returns a handle to the instantiated CLI object.	The <code>new</code> operator returns a pointer to the instantiated native object on the native heap.
It is not mandatory to delete handles, the Garbage Collector will eventually clean up all orphaned managed objects.	It's your responsibility to call <code>delete</code> on pointers to objects that you have allocated; if not, you will suffer a memory leak.
Handles cannot be converted to and from a <code>void^</code>	Pointers can convert to and from a <code>void*</code>
Handles do not allow handle arithmetic.	Pointer arithmetic is a pretty popular mechanism to manipulate native data, especially arrays.

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

Now that we have 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 compliment handles. The standard C++ reference cannot obviously be used with a managed object on the CLR heap as it's not guaranteed to remain in the same memory address for any period of time. So the tracking reference had to be introduced and as the name suggests it tracks a managed object on the CLR heap, so that even if the object is moved around by the GC, the tracking reference will still hold a reference to it. Let's look at a function that accepts a `String^` argument and then assigns a string to it. Our first version will not work as expected, and the calling code will find that the `String` object it passed to the function has not got 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 the above 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 will now see that `str` does get changed, because the function takes a tracking reference 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`. So, whenever you want to pass a CLI handle to a function, and you expect the handle itself to be changed within the function, then 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. Possibly 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 on using 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 will always expect a handle to an object (since C++ is the only language that supports a non-handle 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^
```

Something to be aware of is that the `*` punctuator is used to dereference both pointers and handles, though symmetrically thinking, a `^` punctuator should have been used to dereference a handle. Perhaps, the reason this was designed this way is to allow us to write agnostic template/generic classes that would work on both native and unmanaged types.

Alright, now we know how to declare a CLI type and we also know how to use handles to a CLI type. To put these skills to use, we'll need to understand how CLI types are instantiated which is exactly what we are going to do in the next section.

1.5 Instantiating CLI classes

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

1.5.1 The `gcnew` operator

The `gcnew` operator is used to instantiate CLI objects and it returns a handle to the newly created object on the CLR heap. While quite similar to the `new` operator, there are some important differences, namely that `gcnew` does not have either an array form or a placement form, and it cannot be overloaded either globally or specifically to a class. A placement form would not make a lot of sense for a CLI type when you consider that the memory is allocated by the Garbage Collector, and it's for the same reason that you are not permitted to overload the `gcnew` operator. The reason there's no array form for `gcnew` is that CLI arrays use an entirely different syntax from native arrays, which we'll cover in detail in the next chapter. If the CLR cannot allocate enough memory for creating the object, a `System::OutOfMemoryException` is thrown, though chances are pretty low that you would ever run into that situation. If you do get an `OutOfMemoryException`, and your system is not running low on virtual memory, it would most likely be due to some badly written code such as an infinite loop which keeps creating objects or a stack overflow caused due to an incorrectly coded recursive function. 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, though the C++/CLI compiler uses a different mechanism to handle the usage of the `gcnew` operator to create value type objects (which I will describe later in this section). Since `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, it's `newobj` that throws `System::OutOfMemoryException` when it cannot 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 does not successfully complete, as would be the case if an exception was raised inside the constructor, `gcnew` will not return a handle, and 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 very surprisingly, you will see that `student` is still `nullptr` when it executes the `if`-block. Since the constructor did not complete executing, the CLR concludes that the object has not fully initialized and it does not push the handle reference on the stack (as it would have had the constructor completed successfully).

[Notes] `nullptr`

C++/CLI introduces the concept of a universal null literal called `nullptr`. This allows us to 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 zero as dictated by standard C++ and 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's different from what's generated when you instantiate a `ref` type. For example, consider the following code where we use `gcnew` to instantiate a value type.

```
value class Marks
```

```

{
public:
    int Math;
    int Physics;
    int Chemistry;
};

//...

Marks^ marks = gcnew Marks();

```

For the above code, the C++/CLI compiler uses the `initobj` MSIL instruction to create a `Marks` object on the stack, and this object is then boxed to a `Marks^` object. We will discuss boxing and unboxing in the next section, but for now, note that unless it's imperative to the context of your code to `gcnew` a value type object, doing so is pretty inefficient. A stack object has to be created and this has to be boxed to a reference object – so not only do you end up creating two objects, 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;
```

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

1.5.2 Constructors

If you have a `ref` class and you have not 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's generated for you will call the constructor of the immediate base class for the current class and if you haven't specified a base class, it calls the `System::Object` constructor as every `ref` object implicitly derives from `System::Object`. For example, consider the following two classes, both of which do not have user defined constructors.

```

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

```

Both `Student` and `StudentBase` in the preceding snippet do not have 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` will call the constructor for the `StudentBase` object:

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

And the generated constructor for `StudentBase` will call the `System::Object` constructor:

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

Please post comments or corrections to the Author Online forum at www.manning.com/sivakumar

Just as with standard C++, if you have a constructor, either a default constructor or one that takes one or more arguments, the compiler will not 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, will initialize the static members of a class. Static constructors cannot have parameters, and 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 those special methods have a `.` in their names is that this avoids name clashes, because none of the CLI languages allow the use of 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. So, when you have a simple class such as the one below, the generated MSIL will have a static constructor, even though you haven't specified one on your own.

```
ref class StudentBase
{
    static int number;
};
```

The generated class looks more like this due to the compiler-generated constructors and the implicit derivation from `System::Object`.

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

A value type cannot declare a default constructor because the CLR cannot guarantee that any default constructors on value types will get called appropriately, although members are zero-initialized automatically by the CLR. In any case, a value type should be a simple type that exhibits value semantics, so it shouldn't really 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 cannot have user defined destructors, copy constructors and copy assignment operators.

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

I guess 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, specially when you design complex class hierarchies. And as we

progress through this book and see more examples, you should feel a lot more comfortable with such differences.

Now, I'd like to discuss copy constructors, since we've already talked about constructors.

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. But this is not the case for managed classes. Consider the following bit of code where we attempt 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 will get compiler error C3673 (*class does not have a copy-constructor*). The reason for this error is that unlike in standard C++, the compiler will not generate a default copy constructor for your class. At least one reason why this is so is that all `ref` objects implicitly derive from `System::Object` which does not have a copy-constructor; so even if the compiler attempted to generate a copy constructor for a `ref` type, since it wouldn't be able to access the base class copy constructor (it doesn't exist), it would fail.

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 would fail because the base class copy constructor is inaccessible. To demonstrate that, let's write a class that's 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
```

The above code will not compile, because the compiler is unable to copy construct the derived object, since the base object's copy constructor is declared as `private`, and therefore inaccessible from the derived object. What happens with a `ref` class is pretty much similar to this code. In addition, unlike

native C++ objects which are not polymorphic unless you access them via a pointer, `ref` objects are implicitly polymorphic (since they are always accessed via reference handles to the CLR heap), which means that a compiler generated copy constructor may not always do what you expect it to do. And 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, and the VC++ team presumably decided that there are too many equations there to have the compiler generate copy constructors for classes that don't define them.

So, if you want copy-construction support for your class, you are going to have to implement it explicitly, which fortunately is not 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 I had to explicitly add a default parameter-less constructor to the class; that's because it won't get generated by the compiler when it (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 alright except that you may have a `Student` object that you want to pass to the copy constructor. If you are wondering how that's even possible, C++/CLI supports stack semantics and we will cover this in detail in the next chapter. Anyway, assume that we have a `Student` object `s1` instead of a `Student^` and we need to use that to invoke a copy constructor:

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

Now, that won't even compile. There are two ways to get that to compile, the first 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);
```

While that compiled and solved the immediate problem, it's not a complete solution when you consider that every caller of your code needs to do the same thing if what they have is a `Student` object instead of a `Student^`. An alternate solution is to have two overloads for the copy constructor, as shown in the following listing.

Listing 1.2 Declaring 2 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);

```

While this does solve the issue of a caller requiring to have the right form of the object, it brings with it another problem of its own, namely code duplication. You could wrap the common code in a private method and have both overloads of the copy constructor call this method, but this means you cannot take advantage of initialization lists.

Eventually, it's a design choice you'd have to make. [#1] If you only have the copy constructor overload taking a `Student^`, then you'd need to use the unary `%` operator when you have a `Student` object, and [#2] if you only have the overload taking a `Student%`, then you'd need to dereference a `Student^` using the `*` operator before using it in copy-construction. And 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 would be to use the overload that takes a handle (in our example, that would be the one that takes a `Student^`), because this overload is visible to other CLI languages like C# (unlike the other overload), which is a good thing if you ever run into language interop situations. The unary `%` operator is not really going to slow down your code and when you think of it, it's just an extra character that you need to type. I'd also suggest that you stay away from using two overloads, unless it's a very specific case of a library that will be exclusively used by C++ callers and even then, there's still the issue of code duplication to think of.

Alright, now you know that if you need copy construction on your ref types, you are going to have to implement it yourself, so it might not be very surprising when we 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 will generate automatically for native classes in standard C++, but this is not so for a `ref` class and the reasons for this are pretty similar to those that dictated that a copy constructor will not be automatically generated. The following code (that uses the `Student` class we defined earlier) will not 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 would do in standard C++, except that the types are now managed :

```

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

```

Note that the copy assignment operator can only be used by C++ callers, since it would be 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 in as many of the good C++ programming practices you followed into the CLI world, except where they are not applicable. As an example, our assignment operator does not handle self-assignment. While, in our specific example, it does not matter, consider the case below :

Listing 1.3 The self-assignment problem

```
ref class Grades [#1]
{
    //...
};

ref class Student
{
    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; [#2]
        m_grades = s.m_grades;
        return *this;
    }
    void SetGrades(Grades^ grades)
    {
        //...
    }
};
```

(#1) : <Class with non-trivial ctor/dtor>

(#2) : <Possible problem if self assignment occurs>

In the preceding listing, [#1] assume that `Grades` is a class with a non-trivial constructor and destructor; thus, in the `Student` class assignment operator, before the `m_grades` member is copied, [#2] the existing `Grades` object is explicitly disposed by calling `delete` on it, which is all very efficient. Now assume that it so happens 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 , we end up with a corrupted `Student` object with an invalid `m_grades` member. So, just as you would have done in standard C++, check for self-assignment :

```
Student% operator=(const Student% s)
{
    if(%s == this) [#1]
    {
        return *this; [#2]
    }
    m_name = s.m_name;
    if(m_grades)
        delete m_grades;
    m_grades = s.m_grades;
    return *this;
}
```

(#1) : <check for self-assignment>

(#2) : <if it is so, return immediately>

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

1.6 Boxing and Unboxing

Boxing is the conversion of a value type V to an object of type V^{\wedge} 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 $Object^{\wedge}$ or a V^{\wedge} is cast back to the original value type V . Boxing is an implicit process (though it can be explicitly forced too), while unboxing is always an explicit process. If it sounds confusing to you, visualize a real box into which you put some object (say a camera) so that you can send it via FedEx to your friend in the next city – it's pretty much the same that happens in CLR boxing. And when your friend receives the package, he opens the box, and retrieves the camera, which is analogous to CLR unboxing.

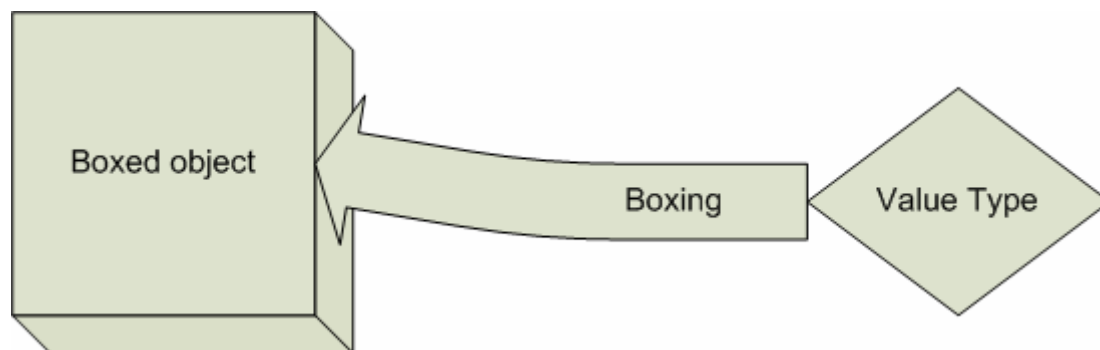


Figure 1.4 The boxing process

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` will be boxed to the CLR heap, and it's this boxed copy that's used by the method. The reason for this is that `ref` types are always references to whole objects on the CLR heap, while `value` types are typically on the stack or even on the native C++ heap. So when a method expects an `Object` reference, the `value` type has to be copied to the CLR heap where it has to behave like a regular `ref` type object. In the same way, when the underlying `value` type has to be retrieved, it has to 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, and several programmers complained about the extra amount of typing required. And since most people felt that the double-underscored keywords were pretty repulsive, the fact that they had to use one of them a gratuitous number of times in the course of everyday programming, made them all that more upset. And you can't really blame them as the following two code listings will show.

Table 1.5 Boxing differences between the old and new syntaxes

Listing 1.4 Explicit boxing in the old MC++ syntax	Listing 1.5 Implicit boxing in C++/CLI
<pre>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));</pre>	<pre>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);</pre>

I guess it would be an understatement if I said that the second code listing is a lot more pleasing to the eye and involves a lot lesser typing. But, implicit boxing has a dangerous disadvantage, which is that it hides the boxing costs involved from the programmer, which can be a pretty bad thing. Boxing is an expensive operation; a new object has to be created on the CLR heap and the value type has to be bit-wise copied into this object. Similarly, whenever there's a lot of boxing involved, chances are good that there's also quite a good bit of unboxing being performed. Unboxing typically involves constructing the original value type and bit-wise copying its data from the boxed object. So as a developer, if you ignore the costs of repeated boxing/unboxing operations, either knowingly or because you didn't realize it, you might run into performance issues, specially 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. So changes in one of them will not be reflected in the other. Consider the following code snippet where we have an `int`, a boxed object containing the `int`, and a second `int` that has been explicitly unboxed from the boxed object. The output of the code will show that, they are three different entities.

```
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 are 3 different entities – the original value type, the boxed type and the unboxed value type. Notice how we had to `safe_cast` the `Object^` to an `int^` before dereferencing it. This is because dereferencing is always done on the boxed value type, so to get an `int`, you have to apply the dereference operator on an `int^`, and hence the cast.

[Notes] `safe_cast`

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 and if so, does the conversion, else it throws a `System::InvalidCastException` exception.

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 will get an `InvalidCastException`. This ensures type-safety when you perform boxing and unboxing operations. Consider the following code listing that demonstrates what happens when you attempt to unbox objects to the wrong value types.

Listing 1.6 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
}
```

Please post comments or corrections to the Author Online forum at www.manning.com/sivakumar

```

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);
}

```

In the above code listing, I have attempted to unbox a boxed double to an int and a boxed int to a double. While 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'.

```

1.6.3 Implementation at the MSIL level

MSIL uses the `box` instruction to perform boxing. Here's 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 :

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 are not going to decipher each MSIL instruction there, but the line of code that is of interest to us 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 bit-wise 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 1st position (the `Object^` variable `o`).

Let's now look at how unboxing is done at the MSIL level:

Table 1.7 MSIL generated for unboxing

C++/CLI code	Generated MSIL
	.locals init ([0] int32 x, [1] object o)
<code>int x = *safe_cast<int^>(o);</code>	IL_0000: ldloc.1 IL_0001: castclass int32 IL_0006: unbox int32 IL_000b: ldind.i4 IL_000c: stloc.0

Unboxing is pretty much the reverse process. The `Object` to be unboxed is pushed on the stack and a cast to `int^` is performed using the `castclass` instruction, and on successful completion of this call, the `Object` on the stack will be of type `int^`. Now, 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 boxing where a new object is created, but `unbox` does not 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` and then copying the value type into this `Object`, and 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 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.

Again, you waste CPU cycles in checking whether the cast is valid and also in the dereferencing operation, which may turn out to be expensive in terms of wasted CPU time, depending on the context of the executing code. Thus, both boxing and unboxing are very expensive operations and while you probably won't see much of a performance decrease for simple applications, bigger and more complex applications may seriously be affected by performance loss if you don't restrict the number of boxing/unboxing operations that are performed. Since 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 cannot initialize a boxed value type to null by assigning a 0 to it. You have to use the `nullptr` constant to do 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
```

Please post comments or corrections to the Author Online forum at www.manning.com/sivakumar

In the second case, the 0 is treated as an `int` which is boxed to an `int^`, so you specifically need to use `nullptr` if you want to assign a handle to null.

When you have two overloads for a function which differs 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. So, you need to keep in mind that, when selecting the best overload, the compiler gives lowest priority to one that requires boxing. In fact, if you had another overload that required a non-boxing cast, that overload would receive preference over the one that requires boxing. So, 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'd suggest you always consciously keep track of the amount of boxing that's being done in your code. Due to its being an implicit operation, you might miss out on intensive boxing operations, but where there's boxing, there's bound to be some unboxing too, so if you find yourself having to do a lot of unboxing, review your code to see if there is an overuse of boxing and try and redesign your class to reduce the amount of boxing/unboxing 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, and as you might have inferred by now, the basic programming concepts remain the same in C++/CLI as in standard C++, but you need to accommodate for the CLI and everything that comes with it like Garbage Collection, handles to the CLR heap, tracking references and the implicit derivation from `System::Object`. Topics we 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 there had to be some changes made due to the nature of the CLI (which is a different environment from native C++), and as long as you are aware of those differences and write code accordingly, you are perfectly alright. While 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 will 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.