Pro Visual C++/CLI and the .NET 3.5 Platform

ISBN-13 (pbk): 978-1-4302-1053-5

ISBN-13 (electronic): 978-1-4302-1054-2

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

Lead Editor: Matthew Moodie
Technical Reviewer: Don Reamy
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Sofia Marchant
Copy Editor: Liz Welch
Associate Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Susan Glinert Stevens
Proofreader: Linda Seifert, Lisa Hamilton
Indexer: John Collin
Artist: April Milne
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

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

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

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

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

CHAPTER 1

Overview of the .NET Framework

First off, let's get one thing straight. This book is about developing code within the confines of the Microsoft .NET Framework 3.5. Therefore, it only makes sense that you start by getting acquainted with the underlying architecture with which you will be developing your code: the .NET Framework.

I cover a lot of material in this chapter, mostly at the 30,000-foot level. The main goal here isn't to make you a .NET expert. This chapter is designed to provide you with a level playing field from which to start your C++/CLI code development while exploring this book.

I start with a brief description of .NET and the .NET Framework and why we programmers need it. Then, I briefly examine the assembly, which is the central building block for all .NET Framework application distribution and execution. Next, I move on to the core of the .NET Framework: the common language runtime (CLR), the common type system (CTS), and the common language specification (CLS). Finally, I discuss, at a very high level, the software components available to .NET Framework developers.

What Is .NET?

I guess getting the definition from the horse's mouth would be a good place to start. Microsoft describes .NET on their Web site (`http://www.microsoft.com/net/Overview.aspx`) in the following way:

> *The .NET Framework is a development and execution environment that allows different programming languages and libraries to work together seamlessly to create Windows-based applications that are easier to build, manage, deploy, and integrate with other networked systems.*
>
> *Built on Web service standards, .NET enables both new and existing personal and business applications to connect with software and services across platforms, applications, and programming languages. These connections give users access to key information, whenever and wherever you need it.*
>
> *Microsoft .NET–connected software makes the "real-time" enterprise real by enabling information to flow freely throughout the organization, accessible to business partners, and delivering value to customers. With .NET-connected software, users can increase the value of existing systems and seamlessly extend those systems to partners, suppliers, and customers.*

Quite a mouthful, don't you think? So what does it mean?

The first thing many developers mistakenly assume is that .NET is strictly a network or Web architecture. You would think so with Microsoft's definition. Heck, even the name ".NET" suggests it. Well, truthfully, .NET sort of is and sort of isn't.

Within .NET are many features that enable a developer to create some truly awesome stand-alone applications—and very easily, I might add. But, according to Microsoft, as their definition suggests, developing stand-alone applications is not the goal of .NET.

That being said, what is .NET? Well, my definition is a little less verbose:

> *.NET is a set of technologies that allow entire software applications to be created rapidly and easily using an integrated network-centric architecture.*

I have to admit that Microsoft's definition does sound much more impressive. But when you boil down Microsoft's marketing fluff, this is really all they are saying.

The key concept Microsoft is trying to push with .NET is interconnectivity between computer systems. True, interconnectivity is hardly new. A host of technologies, such as DCOM, COM+, and CORBA, have been doing this for quite a long time. What make .NET special is how nearly effortless it is to develop this interconnectivity within your applications.

When architecting, designing, and developing using .NET, you are not restricted to your single workstation, LAN, or even your company's WAN. With .NET, your application can use the entire Internet. In fact, not all the parts of your system have to be owned or maintained by your company. What this means is you can have part of your application running in your data center in India, another part in China owned by a third party, which prints out to a client in Russia, and it's all driven from a workstation in the United States. (Okay, lag might be an issue with all these distance places, but that is a hardware issue so it's not my concern… I'm joking… really.)

What is really cool is that .NET uses a technology called the Web service, which is based on XML and allows .NET to interconnect with systems on architectures not based on .NET. Thus, not only can your application be dispersed all over the globe, but the applications it can interconnect with can be Unix, Linux, Mac OS, or any other operating system that supports XML (off the top of my head, I can't think of any).

You might be asking why is this book so large then, if .NET is all about network interconnectivity? This is where the other key concept of my definition comes into play: "entire." True, you are developing network-centric applications, but you are also creating all parts of the application. This means with .NET you can create the presentation tier, business tier, database tier, and anything in between—and in fact you frequently do. To accomplish this, .NET provides a huge framework from which to do your development called the .NET Framework.

■ **Note** Wherever you read the word "Internet," you can assume "intranet" and "extranet" apply as well.

What Is the .NET Framework?

The .NET Framework comprises all the pieces needed to develop, deploy, and execute Web services, Web applications, Windows services, Windows applications, and console applications. (Well, almost all the pieces. IIS is needed for Web services and Web applications.) I discuss each of these in more detail later in the chapter. You can think of the .NET Framework as a three-level hierarchy consisting of the following:

- Application development technologies like ASP.NET, Windows Forms, ADO.NET, Windows Presentation Foundation, Windows Communication Foundation, Windows Workflow Foundation, Windows CardSpace, and LINQ.
- .NET Framework base class library
- CLR

This hierarchy is illustrated in Figure 1-1.

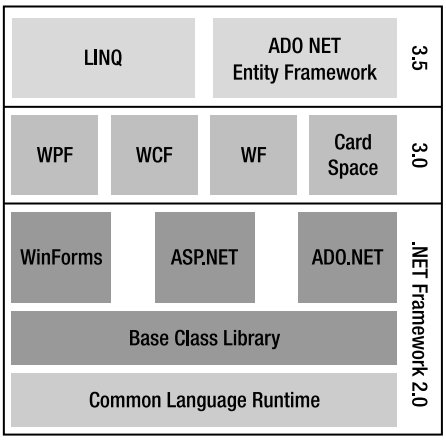

Figure 1-1. *The .NET Framework hierarchy*

Each of the layers in Figure 1-1 is dependent on the layer beneath it. The CLR lies just above the operating system and insulates the programmer from its intricacies. The CLR is what actually loads, verifies, and executes Web services, ASP.NET applications, Windows services, Windows applications, and console applications.

The .NET Framework base classes are a large number of classes broken up by namespaces containing all the predeveloped functionality of .NET. They contain classes to handle things such as file input/output (I/O), database access, security, threading, graphical user interfaces, and so on. As a C++/CLI developer, you will spend many hours perusing and using these classes.

The application development technologies provide a higher layer of abstraction than the base classes. C++/CLI developers will use these technologies to build their Web applications, Web services, and Windows applications. Most of the functionality a developer needs can be found at this level of abstraction, but in those cases where more control is needed, the developer can dive down into the base classes level.

.NET Programming Advantages

The .NET Framework was designed and developed from day one to be Internet aware and Internet enabled. It uses technologies such as SOAP and XML as its underlying methods of communication. As a developer, you have the option of probing as deeply as you wish into each of these technologies, but with the .NET Framework, you have the luxury, if you want, of staying completely ignorant of them.

You have probably heard that .NET is language neutral. This key feature of .NET is handled by .NET compilers. The reason this is possible is because all the .NET compilers compile to Microsoft Intermediate Language, better known as MSIL or just IL. It is currently possible to develop code using the languages provided by Microsoft, (C++/CLI, C#, J#, JScript .NET, and Visual Basic 2008) or in one of the many other languages provided by third parties (such as COBOL, Delphi, and Perl). All .NET-compatible languages have full access to the .NET Framework base class library. I cover .NET multilanguage support briefly in this chapter.

Another thing you have probably heard whispers about is that .NET can be platform independent. Okay, that is not entirely accurate, but the underlying CLR and CLI (you will read about these shortly) is an ECMA standard that can be implemented on multiple platforms. This means that it is possible to port the .NET Framework to non-Windows platforms and then run it without recompiling

.NET applications. The reason for this is that .NET-compatible code is compiled into something called *assemblies*, which contain code, along with several other things, in an intermediate language. I cover assemblies briefly in this chapter and then delve into the art of working with them in Chapter 20.

■ **Note** It is true that the .NET Framework can be ported. Two such ports, Mono and DOTGNU, for the Linux platform are probably the best-known ports of the .NET Framework. Microsoft has also provided Rotor for multiple platforms such as MAC and BSD Unix.

If you've been coding and deploying Windows code in C++ for any length of time, I'm sure you've become painfully aware that it's anything but simple. Now, if you've gone beyond this to build distributed applications, the complexity is multiplied many times over. A key design goal of the .NET Framework is to dramatically simplify software development and deployment. Some of the most obvious ways that the .NET Framework does this are as follows:

- It usually shelters you from the complexities of the raw Windows application programming interface (API). However, there are several APIs in Win32 that have not being implemented in .NET and still require the use of P/Invoke to gain access. I cover P/Invoke in Chapter 23.
- It provides a consistent, well-documented framework, and with it, users can create their own consistent self-documented frameworks and applications. You will see how to do this with integrated XML documentation covered in Chapter 6.
- Managed code is used to create objects that can be garbage collected. You no longer have to worry about memory loss because you forgot to delete allocated pointers. If you use managed code, you don't have to deallocate pointers because the .NET Framework does not use pointers; instead, it uses handles, and the .NET Framework does the deleting of allocated memory for you. (Okay, reality check: the fact is that occasionally memory loss does happen, but it is a very rare occurrence.)
- The intricacies of COM and COM+ have been removed. To be more accurate, COM and COM+ are not part of the .NET Framework. You can continue to use these technologies, but .NET supports them by placing COM and COM+ components in a class library–derived wrapper. You no longer have to worry about things such as the VARIANT, IUnknown, IDL, and so on.
- Deployment components no longer use the registry or special directories.
- Deployment is frequently as simple as an xcopy.

A Closer Look at the .NET Framework

Okay, you have looked at .NET and the .NET Framework in general terms. Now, let's break it into the elements that are relevant to a C++/CLI programmer and then look at each element in some detail. There are five major elements that a C++/CLI developer should have at least a basic knowledge of before attempting to code. Each element affects the C++/CLI programmer differently:

- *Assemblies*: A form of binary distribution
- *CLR*: A way of executing
- *CTS*: A way of defining data-storage types
- *CLS*: A specification of language-neutral support
- *.NET Framework base class library*: A whole set of development objects to learn

I discuss each of these elements in more detail in the following sections.

Assemblies

You need a basic understanding of assemblies (see Figure 1-2) before you can learn about any other element of the .NET Framework. I cover some basic information about assemblies in this chapter and then discuss working with them in detail in Chapter 20.

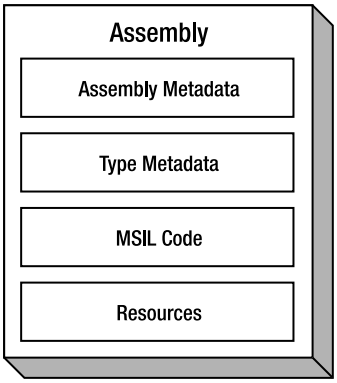

Figure 1-2. *The basic assembly structure*

Assemblies are the core building blocks for all .NET Framework application distribution and execution. They are generated after compiling C++/CLI code. Like pre-.NET application deliverables, they end with either `.exe` or `.dll`, but that is pretty well as far as the similarities go.

Basic Structure

Assemblies are a self-describing collection of functionalities stored in an intermediate language and/or resources needed to execute some portion of an application. Assemblies are made up of four sections:

- The assembly metadata
- Type metadata
- Microsoft Intermediate Language (MSIL) code
- Resources (Figure 1-2)

All sections except the assembly metadata are optional, though an assembly made up of just assembly metadata sections won't do anything.

Assemblies can be either private or shared. *Private assemblies* reside in the same directory as the application itself or in one of its child directories. *Shared assemblies*, on the other hand, are stored in the global assembly cache (GAC). The GAC is nothing more than a directory structure that stores all the assemblies that are globally available to the computer (Figure 1-3). A neat feature of the GAC is that more than one version of the same assembly can reside in it.

A key feature of all assemblies is that they are self-describing. In other words, all information needed to understand how to use the assembly can be found in the assembly itself. An assembly does this by including metadata directly within itself. An assembly has two different metadata sections: the assembly metadata and the type metadata. You gain access to this metadata using reflection, which I cover in Chapter 20.

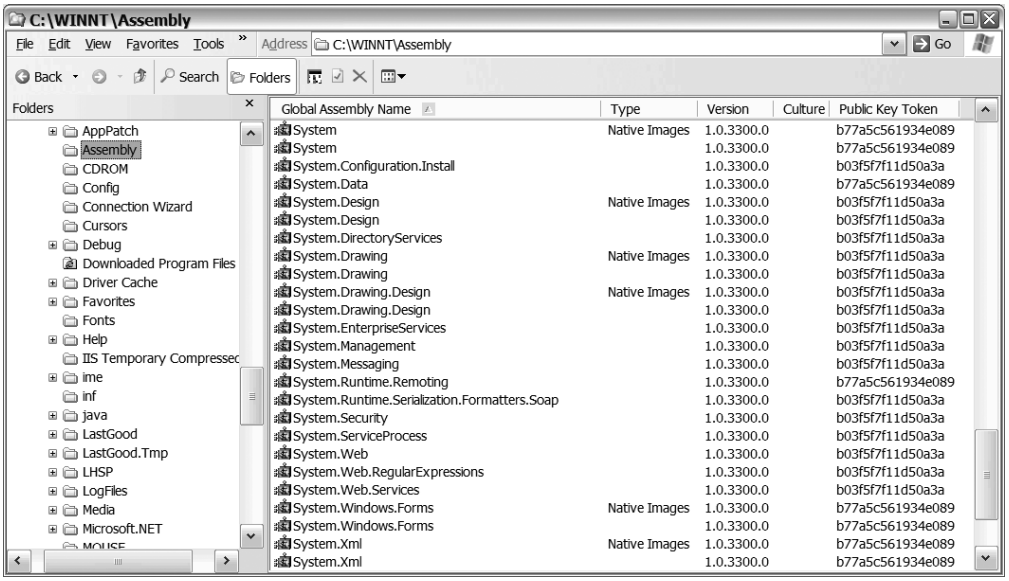

Figure 1-3. *The global assembly cache*

Metadata

The *assembly metadata* is also known as the *assembly manifest*. As its name suggests, the assembly metadata describes the assembly. Here is a list of some of the assembly metadata's contents:

- The name of the assembly.
- The version number.
- The culture used by the assembly (in other words, localizable information such as language, currency, number formatting, and so on).
- Public key and digital signature. These provide a uniquely identifiable ID of who created the assembly.
- A list of all files that make up the assembly.
- A list of all referenced assemblies.
- Reference information for all exported classes, methods, properties, and so on, found in the assembly.

The *type metadata*, however, describes the types within the assembly. The type metadata generated depends on the type being created. If the type were a method, then the metadata generated would contain things such as the name, return types, number of arguments and their types, and code access security level. A property, on the other hand, would reference the get and set methods; these methods in turn would contain names, return types, and so on.

A nice feature of metadata is that it can be used by many of the tools available to the C++/CLI developer. For example, Visual Studio's IntelliSense statement completion functionality (Figure 1-4) is driven using the reference assembly's metadata and not some secondary description file. Because it comes directly from an assembly, IntelliSense will also work for assemblies you have written yourself without any additional effort on your part.

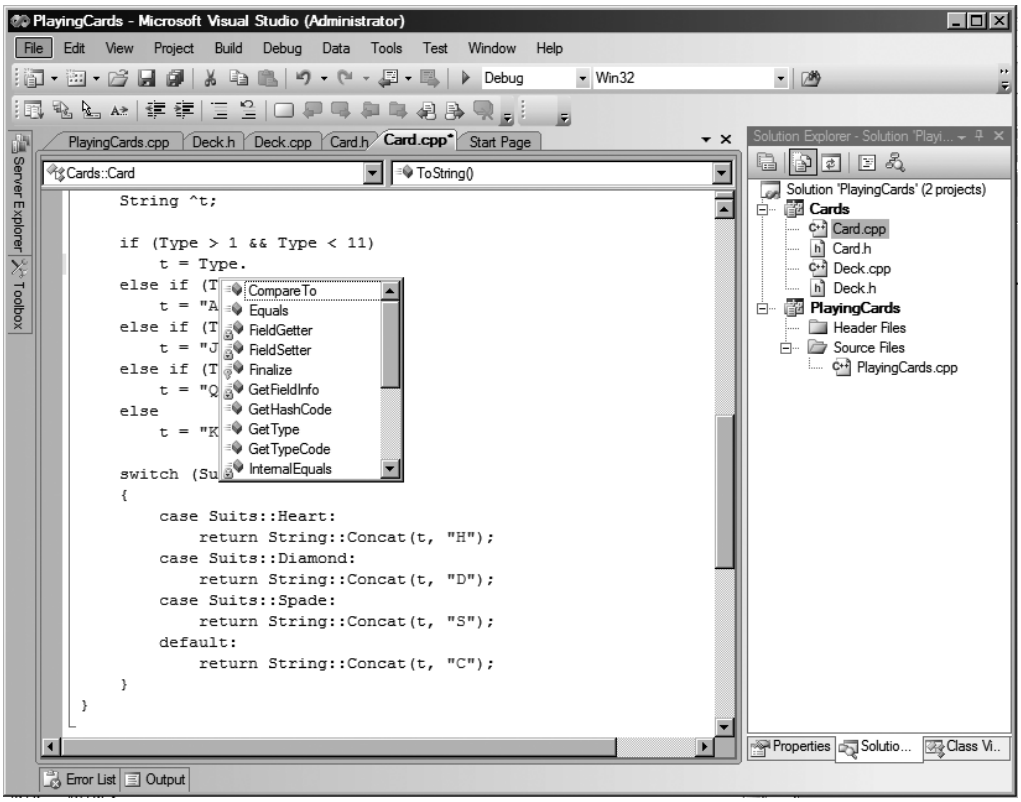

Figure 1-4. *Visual Studio's IntelliSense using metadata*

Versioning

Application assemblies are very version-aware when they're referencing strong-named assemblies within the GAC. Every assembly has a version number. Also, every referencing assembly stores the version number of any assembly that it references. It's not until the referenced assembly is strong named and in the GAC that the referencing assembly automatically checks when executing, via the CLR, that the versions match before it continues to execute. I cover assembly versioning in detail in Chapter 20.

Microsoft Intermediate Language

A major change from standard C++ compilation that is hidden for the most part under the covers but that you should be aware of as a C++/CLI programmer is that C++/CLI code gets compiled to MSIL and not machine code. MSIL is a CPU-independent set of instructions similar to an assembly language. For example, it contains arithmetic and logical operators and flow control. But, unlike the average assembly language, it also contains higher-level instructions to load, store, initialize, and call class objects.

Just for some grins and giggles, here is an example of some MSIL generated from a simple C++/CLI program. See if you can figure out what it does.

```
IL_0000: ldarg.0
IL_0001: ldarg.1
IL_0002: add
IL_0003: stloc.0
IL_0004: ldstr "{0} + {1} = {2}"
IL_0009: ldarga.s val1
IL_000b: call instance string [mscorlib]System.Int32::ToString()
IL_0010: ldarga.s val2
IL_0012: call instance string [mscorlib]System.Int32::ToString()
IL_0017: ldloca.s total
IL_0019: call instance string [mscorlib]System.Int32::ToString()
IL_001e: call void [mscorlib]System.Console::WriteLine(string,
object,
object,
object)
IL_0023: ret
```

For those of you who are curious, the preceding code adds two numbers together and then writes the result out to the console.

MSIL is easily converted to native code. In fact, just prior to the MSIL code running, the CLR rapidly compiles it to native code.

■ **Note** The MSIL in an assembly is compiled prior to execution. It is *not* interpreted at runtime.

■ **Note** It is possible to precompile assembles to speed up the initial load time of a .NET application. This is done with a tool called the Native Image Generator, or `NGEN.exe`.

One key characteristic of MSIL is that it is an object orientation–based language with the restriction of single class inheritance, although multiple inheritance of interfaces is allowed. All types, both value and reference, used within the MSIL must conform to the CTS. Any exposed types must follow the CLS. I cover both CTS and CLS later in this chapter. Error handling should be done using exceptions.

MSIL is the key to .NET's capability to be language neutral. All code, no matter what the programming language, is compiled into the same MSIL. Because all languages ultimately compile to the same MSIL, it is now possible for encapsulation, inheritance, polymorphism, exception handling, debugging, and so on, to be language neutral.

MSIL is also one of the keys to .NET's capability to be platform independent. With MSIL, you can have "write once, run anywhere" ability, just as you do with Java. All that is required for an assembly to run on a non-Windows platform is for the ported CLR to compile MSIL into non-Windows-specific code.

With the combination of MSIL and metadata, .NET is capable of providing a high level of security. For example, strong names found in metadata can ensure that only trusted assemblies are run. If you add code verification to this, provided when your code is compiled with the `/clr:safe` option, then the CLR can ensure that only managed code running with valid privileges is executed.

Resources

In .NET, resources (such as string tables, images, and cursors) can be stored in two places: in external `.resources` files or directly within an assembly. Accessing the resources in either location is extremely easy, as the .NET Framework base class library provides three straightforward classes for access within the `System::Resources` namespace. I cover these classes in detail in Chapter 20, but if you want to get a head start and look them up yourself, here they are:

- `ResourceManager`: Used to access resources from within an assembly
- `ResourceWriter`: Used to write resources to an external `.resources` file
- `ResourceReader`: Used to read resources from an external `.resources` file

In addition to these classes, the .NET Framework provides the utility `resgen.exe`, which creates a `.resources` file from a text file containing key/value pairs.

The `resgen.exe` utility is very useful if you wish to make your Windows applications support multiple (human) languages. It's easy to do this. Simply create multiple `.resources` files, one for each language. From these, build satellite assemblies for each language. Then the application will automatically access the correct language resource based on the current culture specified on the computer. You'll learn how to do this in Chapter 20.

Common Language Runtime

Runtimes are hardly a new concept when it comes to code execution. Visual Basic 6.0 has `msvbvm60.dll`, and Java, of course, has the Java Virtual Machine (JVM). The common language runtime (CLR) is .NET's runtime system.

Do we need another runtime? What makes this one that much better than all the rest? It is simply the fact that the CLR is designed to be the runtime for all languages and (possibly) all platforms. Or, in other words, you no longer need a myriad of different runtimes to handle each programming language and platform. Instead, all you need is the CLR.

It's a pretty big claim. Does it hold water?

There are two common roles for runtimes: to execute code and/or to add common functionality used by most applications. The CLR performs both of these roles for the .NET Framework. But these roles are only the tip of the iceberg. The CLR also performs several other services, such as code verification, access security, garbage collection, and exception handling, and it also handles multilanguage support and compiles MSIL into the native language of the platform (Figure 1-5).

Starting up an application in .NET is conceptually very simple. The CLR loads the application assembly, any referenced developer assemblies, and any referenced base class library assemblies. Then, the application is optionally verified for type safety and valid access security. Next, the loaded MSIL, with the help of information found in the metadata from the assemblies, is compiled into native code. Finally, the application is executed.

The CLR was designed to help provide the following:

- *Simplified programming infrastructure*: Much of the low-level plumbing (memory management, local and remote process communication, etc.) is handled automatically or hidden unless access is needed.
- *Scalability*: Areas that allow for scalability (memory management, process communication, component management, and so on) are contained, already optimized, within the framework.
- *Simple, safe deployment*: A simple xcopy is usually all that is required for deployment.

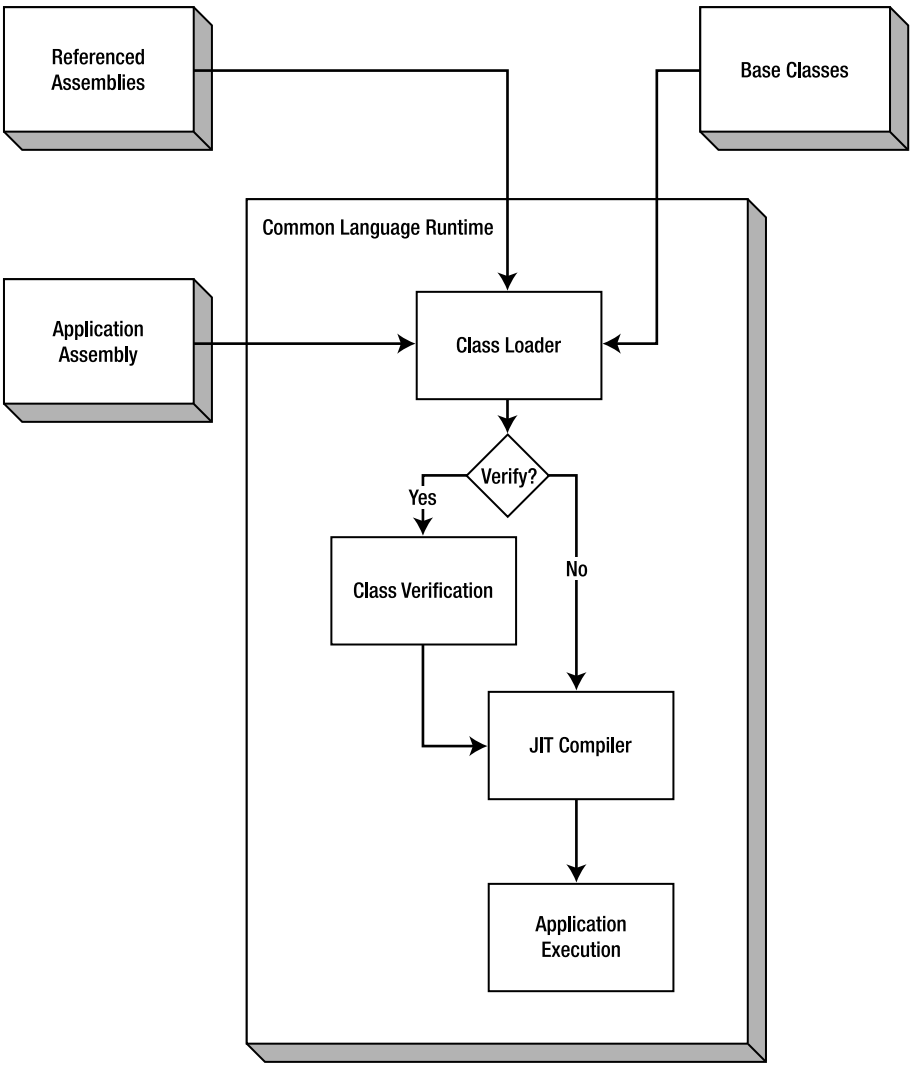

Figure 1-5. *The CLR start-up process flow*

Managed Data

Managed data is data that is allocated by an application and then deallocated by the CLR's garbage collector. All .NET languages except C++/CLI default to managed data. To create managed data in C++/CLI, you must create a reference type within your source code. Chapter 3 will explore how to create managed data in C++/CLI.

Managed Code

Basically, *managed code* is code targeted for the .NET Framework's CLR, which provides things such as memory management, code access security, and multilanguage integration.

Conversely, *native code*, sometimes known (inaccurately) as unmanaged code, is not targeted for the CLR. Native code runs outside of the CLR sandbox, meaning that you lose things like garbage collection and code security. It is possible for the .NET Framework to access C DLLs, COM, and COM+ services, even though all of these are native. I cover native code in detail in Chapters 22 and 23.

Introduced in .NET version 2.0 was the ability to create *safe code*, or managed code that is verifiable by the CLR. *Unsafe code*, as you may expect, is code that can't be verified (Chapter 22 goes into detail explaining what it means for code to be verified). It is usually in the form of native code, but that is not a requirement. Unsafe code can also be compiled to MSIL and run in the CLR (thus, managed code by definition). I cover unsafe code in detail in Part 3. I will also be pointing it out in passing throughout the book.

All the compilers that come with the .NET Framework default to generating managed code except C++/CLI. To create managed code in C++/CLI, you need to add one of the .NET command-line switches when compiling:

- `/clr:oldSyntax`: This switch is used to compile C++/CLI code from .NET versions 1.0 and 1.1. It will generally create a mixed image of native and managed code.
- `/clr`: This switch is used for the C++/CLI code syntax of .NET 2.0 and later. It will generally create a mixed image of native and managed code.
- `/clr:pure`: This switch is used to generate managed code and unmanaged data. If you use unsafe code, the compile will fail.
- `/clr:safe`: This switch is used to generate managed code and managed data. If you use unsafe code and/or unmanaged data, the compile will fail.

When you use Visual Studio, simply select one of the C++/CLI project templates, and these will set the `/clr` switch for you. However, I suggest that you change the switch to `/clr:safe` if you plan to use only managed code, as this book does in most examples.

Common Language Runtime Services

Along with loading and executing an assembly, the CLR provides several other services. Code verification and code access verification are optional services available before the assembly is loaded. Garbage collection, on the other hand, is always active while the assembly is being executed.

Code Verification

The *code verification* service is executed prior to actually running the application. Its goal is to walk through the code, ensuring that it is safe to run. It checks to make sure that the code

- Has not been replaced
- Has not been tampered with since its creation
- Has not been overwritten by another version of the same code

Code Access Verification

Code access verification also walks through the code and checks that all code has the permission to execute. The goal of this service is to try to stop malicious attacks on the user's computer.

A simplified way of looking at how this service works is that the CLR contains a list of actions that it can grant permission to execute, and the assembly contains a list of all the permissions it requires to run. If the CLR can grant all the permissions, the assembly runs without problems. If, however, the CLR can't grant all the permissions, it runs what it can but generates an exception whenever it tries to do something that it doesn't have permission to do.

Garbage Collection

Garbage collection is the mechanism that allows a runtime to detect and remove managed objects from the managed heap that are no longer being physically accessed within the application. The .NET Framework's garbage collector has the added bonus of compacting the memory after it removes the unused portion, thus keeping the memory footprint of the applications as small as possible. This bonus can complicate things sometimes, as managed objects in .NET do not have a fixed location, but you can overcome this with the `pin_ptr<>` keyword. I cover `pin_ptr<>` in Chapter 22. Also, because managed objects are referenced using handles and not pointers, pointer arithmetic is gone except in unsafe sections of the code.

Garbage collection presents a big change to most C++/CLI programmers, because it means an end to most of those annoying memory leaks that plague them while developing. It also has an added bonus: programmers when dealing with memory management no longer have to figure out where to call the `delete` command to the classes that they've created using the `gcnew` command.

■ **Caution** We will see in Chapter 3 that programmers still have to be aware of when to call the `delete` command if they are working with computer resources.

Garbage collection is not the default for C++/CLI. Because this is the case, there are a few things (covered in Chapter 3) that C++/CLI programmers need to learn before they can use garbage collection—in particular, the keyword `ref`. Fortunately, since version 2.0 of .NET, unlike in prior versions of C++/CLI, programmers have gained control of when a managed object gets deleted.

■ **Note** Well, there actually wasn't a prior version of C++/CLI. What came before C+/CLI was Managed Extensions for C++, or Managed C++. If you want to learn about Managed C++, you can read about it in my book *Managed C++ and .NET Development* (Apress, 2003).

Attributes

Attributes are a way for developers to provide additional information about the classes, methods, or data types to the assemblies they are creating. This additional information is stored within the assembly's metadata.

There are several predefined attributes that the compiler can use to help during the compile process. For example, the `System::Obsolete` attribute causes the compiler to generate a warning when it encounters an obsolete method in a class library assembly.

You will see in Chapter 20 how to work with attributes and how it is possible to add your own custom attributes to the assembly metadata.

All attributes—developer code-created and compiler-generated—can be extracted using reflection.

Reflection

An interesting service provided by the CLR is *reflection*. This is the ability to programmatically examine the metadata within an assembly, including the one executing the reflection code. This service allows access to the metadata information, such as details about classes, methods, properties, and so on, contained within the assembly.

Most likely, you will use reflection mainly to get attribute information out of the assembly metadata. For more advanced C++/CLI developers, reflection provides the ability to extract type information within a class so that they can use it to generate types dynamically.

Reflection is accomplished using the myriad classes in the `System::Reflection` namespace. Chapter 20 covers reflection.

Multiple Language Support

.NET had the ambitious goal of creating a completely language-neutral environment for developing software. Some of the features the .NET Framework and Visual Studio developers had in mind were the following:

- Common data types should be shared by all languages.
- Object handles and/or references from any language should be able to be passed as an argument to a method.
- Calling methods from classes created in other languages should be possible.
- Classes should be able to contain instances of other classes created in a different language.
- Inheriting from classes created in another language should be possible.
- The development and debugging environment for all languages should be the same.

Believe it or not, every one of those features is now supported by the CLR and MSIL.

The idea is to pick the best language for the job. Each language has its strong and weak points when it comes to software development. With language-neutral development, you can select the language that best suits the type of development needed.

Have developers accepted this concept? In this age of computer-language holy wars, it seems a little doubtful. Plus, allowing the use of multiple languages during the development of a project does add complexity. Having said that, though, I've worked on a large project that used C, C++, COBOL, HTML, Macro (Assembler), and SQL, plus an assortment of batch scripting languages. To make things worse, each of these languages had different tools for development, and debugging was a nightmare. I don't even want to talk about passing data between modules created in different languages. What I would have given for .NET back then.

How does the .NET Framework create a language-neutral environment? The key is a combination of MSIL and metadata. Basically, the MSIL code tells the CLR what to do (which commands to execute), and the metadata tells it how to do it (which interfaces to use). For a language to be .NET Framework compliant, it obviously needs to be compiled into MSIL code and metadata and placed in an assembly (Figure 1-6).

Because all languages have this requirement, Microsoft was able to tweak each compiler they developed so that it created code to conform to their MSIL and metadata language-neutral requirements. Also, all languages were changed to conform to the CTS. I cover the CTS later in this chapter.

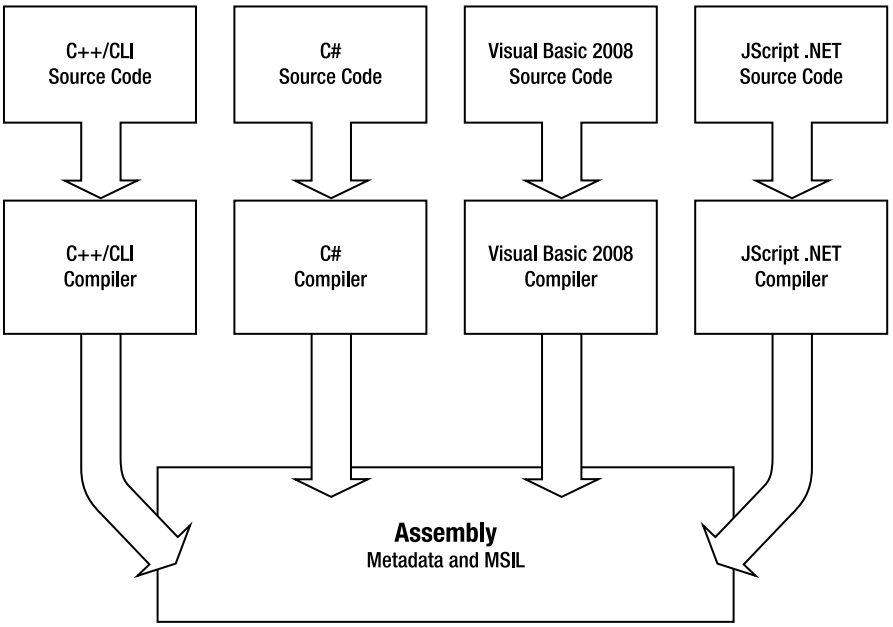

Figure 1-6. *Many compilers, one output*

Multiple Platform Support

By its architecture, the .NET Framework is conducive to multiple platform support. The CLR enables platform independence by providing a runtime layer that sits between the operating system and the application. The just-in-time (JIT) compiler generates machine-specific native code. JIT is covered in the next section, "Just-in-Time Compilation." The MSIL and metadata allow for the "write once, run anywhere" capability that is Java's claim to fame.

Currently, the only multiple platform support provided by the .NET Framework is for Windows-based platforms such as Windows Server 2003, 2008, XP, and Vista. With .NET version 2.0, Microsoft added 64-bit support to the existing 32-bit support, but the operating system is still only Windows based.

What does platform independence mean to C++/CLI programmers? It means a new way of looking at things. C++/CLI programmers think of multiple platform support as coding generically and recompiling on each new platform. With the .NET Framework, developers only need to develop the code and compile it once. The resulting assembly could be run on any supported platform without change.

True, to develop real platform-independent code, developers must use only managed code. If a developer were to use unmanaged code, the assembly generated would become closely coupled with the architecture on which it was compiled.

■ **Note** This book focuses, for the most part, on creating code that is platform independent—though it does delve into unsafe code, which is not platform independent.

Just-in-Time Compilation

Even though .NET applications are stored in an intermediate language, .NET applications are not interpreted. Instead, they are compiled into a native executable. It is the job of the JIT compiler, a key component of the CLR, to convert MSIL code into machine code with the help of metadata found in the executable assembly.

The JIT compiling process is, in concept, very easy. When an application is started, the JIT compiler is called to convert the MSIL code and metadata into machine code. To avoid the potentially slow start-up time caused by compiling the entire application, the JIT compiler only compiles the portions of code that the application calls, when they are called (hence the name, *just-in-time compiler*). After the code is compiled, it is placed in cached memory and then run. The compiled code remains in the cached memory for as long as the application is executing. This way, the portion of code can be grabbed from cached memory, instead of having to go through the compile process each time it is called. There is a bonus in compiling this way. If the code is not called, it is not compiled.

Microsoft claims that managed code should run as fast as native code. How can Microsoft make this claim? The JIT compiler is amazingly fast, but there still is the overhead of having to compile the application each time it is run. This leads you to believe that managed code would be slower.

The key to Microsoft's claim is that JIT compilers generate code specific to the processor type of the machine they are running on. On the other hand, traditional compilers generate code targeting a general range of processor types. For example, the Visual Studio 6.0 C++ compiler generates generic 64-bit machine code. A JIT compiler, knowing that it is run on, let's say, an AMD Athlon 64 X2 Dual-Core Processor, would generate code specific to that processor. The execution time between these two sets of machine code will in many cases be quite different and always in the favor of the JIT compiler-generated code. This increase in speed in the managed code should offset the JIT compiling overhead and, in many cases, make the overall execution faster than the unmanaged code.

Common Type System

The common type system (CTS) defines how all types are declared, used, and managed within the .NET Framework and, more specifically, the CLR. It is also a key component for the CLR's multiple language support. The CTS was designed to perform the following functions:

- Provide an object-oriented data model that can support the data types of all .NET Framework–compatible programming languages
- Provide a set of constraints that the data types of a .NET-compatible language must adhere to so that it can interact with other .NET-compatible programming languages
- Provide a framework for .NET-compatible interlanguage integration and data type safety

Two categories of data types are defined by the CTS: the value type and the reference type. *Value types*, such as `int`, `float`, or `char`, are stored as the representation of the data type itself. *Reference types*, such as handles, classes, or arrays, are stored on the managed heap as references to the location of the data type. You will explore value types and reference types in great detail in Chapter 3.

■ **Note** Unmanaged pointer types are stored on the C Runtime (CRT) heap, which differs from the managed reference type's managed heap.

As you can see in Figure 1-7, all data types fall into one of these two categories.

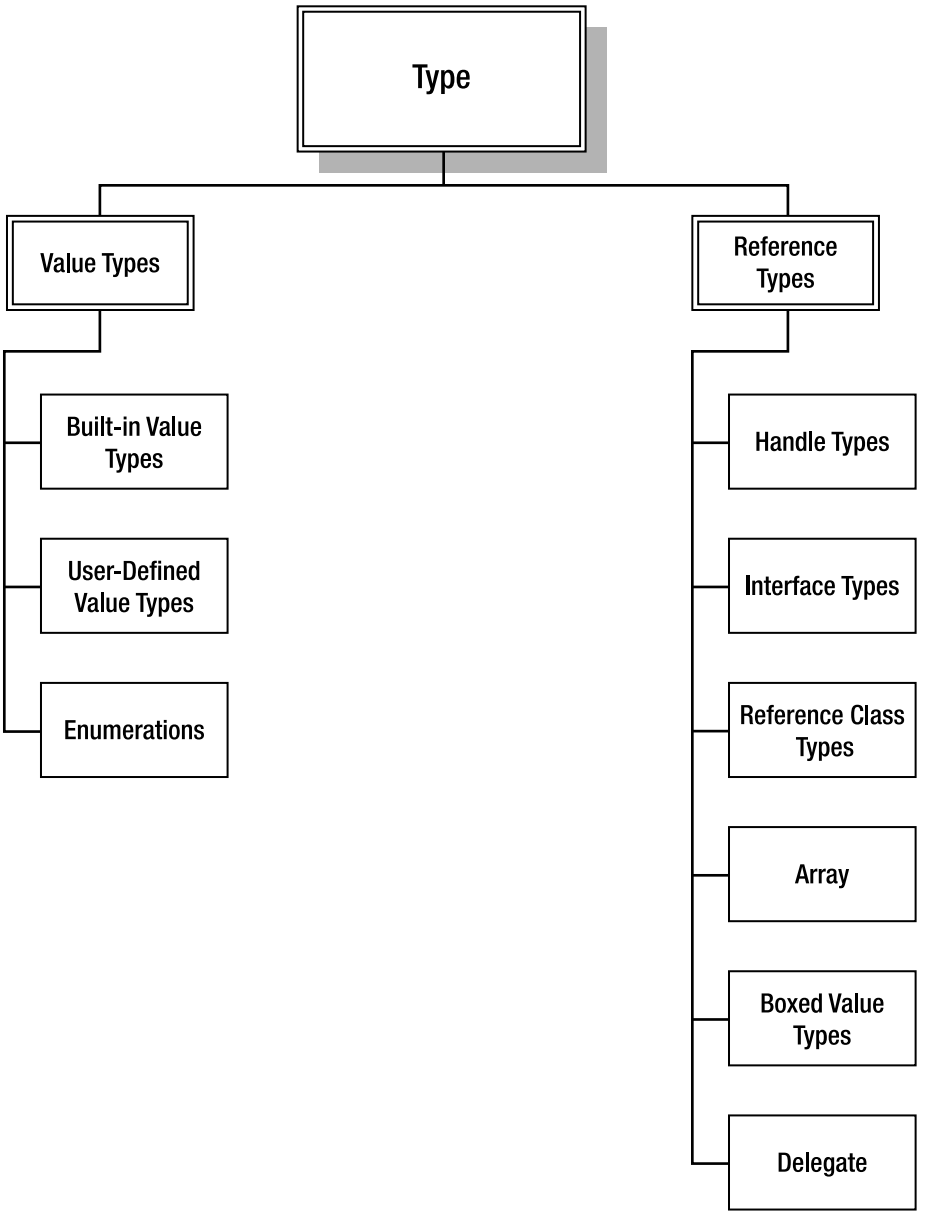

Figure 1-7. *CTS hierarchy*

Let's briefly walk through the hierarchy of all CTS types:

- *Arrays*: A single or multidimensional indexed grouping of types
- *Boxed value types*: A temporary reference to a value type so that it can be placed on the managed heap
- *Built-in value types*: Primitive value types that represent integers, real numbers, Booleans, and characters

- *Reference class types*: A user-defined grouping of types and methods
- *Delegates*: A type that holds a reference to a method
- *Enumerations*: A list of named integer constants
- *Interface types*: A class type where all methods are abstract
- *Handle types*: A reference to a type
- *User-defined value types*: User-defined expansion to the standard, primitive value types

A point worth mentioning is that the CTS defines all .NET-compatible language data types, but a .NET-compatible language does not need to support all CTS-defined data types. In versions prior to .NET 2.0, even Microsoft Visual Basic .NET did not support all data types. This has changed since .NET version 2.0, as you can see in the comparison of the built-in value and reference types supported by Visual Basic 2008, C#, and C++/CLI (Table 1-1).

Table 1-1. *Built-in Value and Reference Types and Their Language Keywords*

Base Class	Visual Basic 2008	C#	C++/CLI
System::Byte	Byte	byte	unsigned char
System::Sbyte	SByte	sbyte	char
System::Int16	Short	short	short or __int16
System::Int32	Integer	int	int, long or __int32
System::Int64	Long	long	long long or __int64
System::UInt16	UShort	ushort	unsigned short or unsigned __int16
System::UInt32	UInteger	uint	unsigned int, unsigned long or unsigned __int32
System::UInt64	ULong	ulong	unsigned long long or unsigned __int64
System::Single	Single	float	float
System::Double	Double	double	double
System::Object	Object	object	Object^
System::Char	Char	char	__wchar_t
System::String	String	string	String^
System::Decimal	Decimal	decimal	Decimal
System::IntPtr	IntPtr	IntPtr	IntPtr
System::UIntPtr	UIntPtr	UIntPtr	UIntPtr
System::Boolean	Boolean	bool	bool

■ **Note** The ^ character in Table 1-1 is not a typo. This is C++/CLI's new handle symbol, which I will cover in Chapter 2.

■ **Caution** You should take care when using `UInt64`, as unpredictable results are possible on Intel 32-bit platforms because they are not thread-safe and do not load the registers atomically.

Common Language Specification

Given that not all of the CTS data types need to be supported by every .NET-compatible language, how then does the .NET Framework maintain that these languages are, in fact, compatible? This is where the common language specification (CLS) comes in. The CLS is a minimum subset of the CTS that all languages must support to be .NET compatible (Figure 1-8).

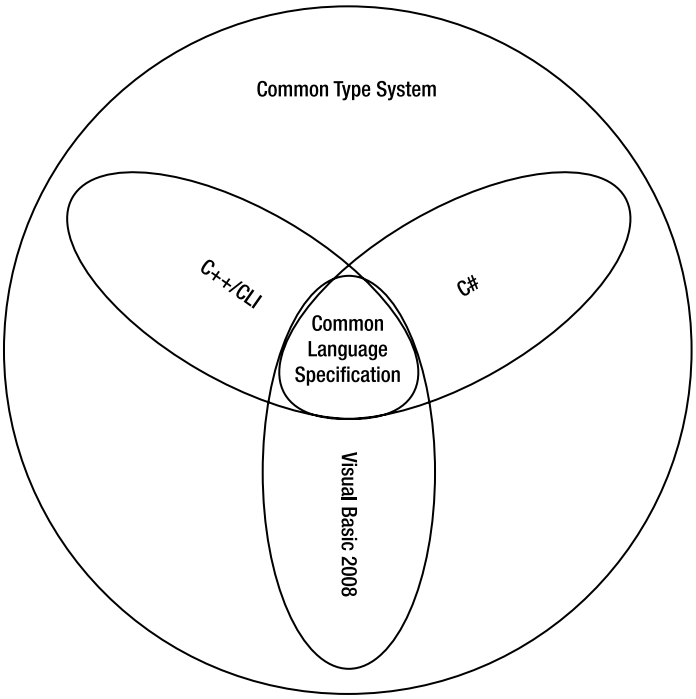

Figure 1-8. *CLS intersection diagram*

To ensure interlanguage operability, it is only the CLS subset that can be exposed by assemblies. Because you can be assured that all languages' building assemblies are using this subset, you can thus also be assured that all languages will be able to interface with it.

■ **Note** When you develop your .NET code, it is completely acceptable to use the entire CTS. It is only exposed types that need to adhere to the CLS for interlanguage operability.

There is no imposed restriction on using the CLS. If you know that your assemblies will only be used by one language, it is perfectly acceptable to use all the types available to that language, even

those that are exposed. Just be aware that if there comes a time when you want to use your assemblies with another language, they may not work because they do not adhere to the CLS.

If you want to view the CLS, you can find it in the .NET documentation. Just search for "What is the common language specification?" The key points that you should be aware of as a C++/CLI programmer are as follows:

- Global methods and variables are not allowed.
- The CLS does not impose case sensitivity, so make sure that all exposed types differ by more than their case.
- The only primitive types allowed are `Byte`, `Int16`, `Int32`, `Int64`, `Single`, `Double`, `Boolean`, `Char`, `Decimal`, `IntPtr`, and `String`.
- Variable-length argument lists are not allowed. Use fixed-length arrays instead.
- Pointers are not allowed.
- Class types must inherit from a CLS-compliant class. `System::Object` is CLS compliant.
- Array elements must be CLS compliant.

Some other requirements might also affect you, if you get fancy with your coding. But you will most likely come across the ones in the previous list.

.NET Application Development Realms

.NET application development falls primarily into one of five realms: Web applications, Web services, Windows applications, Windows services, and console applications. Using languages such as C#, Visual Basic 2008, and C++/CLI in conjunction with Visual Studio provides a simple, powerful, and consistent environment to develop all five. Unfortunately, for C++/CLI, only four are supported completely: console applications, Windows applications, Windows services, and Web services. Web applications, though supported, takes a little more effort.

■ **Note** In the previous version of this book I said that coding a Web application was not possible. I later found out I was wrong and I will show you how in Chapter 17.

Console Applications

Console applications are basically extinct as a final software delivery in the Windows world. Okay, developers and administrators still use them a lot, but the average nontechnical user has GUI-based applications as their final delivery. For developer tools, there is nothing like them. If you need to figure out how to do something, write out a console application. There is nothing simpler and nothing with less overhead. This book is full of console applications for just those reasons.

The key elements of all console applications are the `main()` function and the `System::Console::WriteLine()` method. In fact, that is almost all you need to write a console application.

Windows Applications

Windows applications may be the biggest change for C++/CLI programmers. C++/CLI does not directly support the Microsoft Foundation Class (MFC) library. Wow, don't panic—believe it or not, the .NET Framework has a better solution. It's called Windows Forms, and I'm sure you'll think, as I do, that it's a godsend. With Windows Forms, you get the ease of Visual Basic along with the power

of C++/CLI when you develop Windows applications. I cover Windows applications in Chapters 10 and 11.

■ **Note** Since .NET version 2.0, you can use MFC and Windows Forms somewhat interchangeably, but if you do, the generated assembly will be classified as unsafe.

When you create Windows Forms, you will use the massive `System::Windows::Forms` namespace. Though this namespace is large, it is consistent and well laid out. It will not take you long to get good at using it.

Just to add some variety to your Windows applications, .NET also provides a new and improved Graphical Device Interface (GDI) called, conveniently, GDI+. With GDI+, you can play with fonts, change colors, and, of course, draw pictures. GDI+ is almost worth learning just for one class, `System::Drawing::Image`, which allows an application to load almost any commonly supported graphic file formats, including GIF, JPEG, and BMP, into memory, where they can be manipulated and drawn to the screen. To implement GDI+ in the .NET Framework, you need to explore the `System::Drawing` namespace. I cover GDI+ in Chapter 12.

Web Applications

ASP.NET is a large part of developing Web applications. But unlike traditional Web application development, .NET has changed things. Web applications no longer are run using interpreted scripts. Now they use full-blown compiled applications. These applications are usually written using C# and Visual Basic 2008.

Unfortunately, C++/CLI does not directly support Web applications via Visual Studio. The reason for this is primarily because Microsoft introduced in .NET 2.0 a new construct in C# and Visual Basic called the partial class, which has no equivalent in C++/CLI, and partial classes are heavily relied upon in ASP.NET Web applications. But there are ways around this, as you will see in Chapter 16.

Windows Services

A Windows service is a Windows application that can be started automatically when the operating system boots. However, this is not a requirement, as it is possible to start the Windows service manually.

With the Windows service there is no need for an interactive user or an interface. You will see in Chapter 15 that you do have some limited ability to interface with the service, but to do so, you need a separate control program.

Not only do Windows services not need an interactive user, but they can also continue to run after a user logs off.

Web Services

You might want to think of a Web service as programmable functionality that you execute over the Internet. Talk about remote programming. Using a simple HTTP request, you can execute some functionality on some computer on the opposite side of the world. Okay, there are still some kinks, such as the possible bandwidth problems, but they will be overcome with the current technology advancement rate—that much I am certain of. Chapter 17 covers Web services.

Web services are based on XML technology and, more specifically, the XML-derived Simple Object Access Protocol (SOAP). SOAP was designed to exchange information in a decentralized and

distributed environment using HTTP. For more technical details about SOAP, peruse the World Wide Web Consortium's Web pages on SOAP (`http://www.w3.org/TR/SOAP`).

When you code Web services, you will be working primarily with the `System::Web::Services` namespace. You also get to look at attributes again.

Web services are a key part of Microsoft's plans for .NET because, as you may recall, .NET is about delivering software as a service.

.NET Framework Class Library

Everything you've learned so far is all fine and dandy, but the thing that is most important, and where C++/CLI programmers will spend many a day, is the massive .NET Framework class library. There are literally hundreds of classes and structures contained within a hierarchy of namespaces. C++/CLI programmers will use many of these classes and structures on a regular basis.

With such a large number of elements in the class library, you would think that a programmer could quickly get lost. Fortunately, this is not true. The .NET Framework class library is, in fact, well organized, easy to use, and virtually self-documenting. Namespaces, class names, properties, methods, and variable names usually make perfect sense. The only real exceptions to this that I have found are class library wrapped native classes. I am sure other exceptions exist, but by and large, most namespaces and classes are understandable just by their names. This, obviously, differs considerably from the Win32 API, where obscure names are more the norm.

With the .NET Framework class library, you can have complete control of the computer. That's because the class library functionality ranges from a very high level, such as the `MonthCalendar` class—which displays a single month of a calendar on a Windows Form—down to a very low level, such as the `PowerModeChangedEventHandler`, which notifies the application when the computer is about to be suspended, resumed, or changed from AC to battery, or vice versa.

There are two hierarchies of namespaces in the .NET Framework class library: the platform-neutral `System` namespace and the Microsoft-specific (and aptly named) `Microsoft` namespace. Table 1-2 shows a brief subset of the namespaces that the average C++/CLI programmer will run into.

Table 1-2. *Common .NET Framework Class Library Namespaces*

Namespace	Description
`Microsoft::win32`	Contains classes to handle events raised by the operating system and to manipulate the system registry.
`System`	Contains classes that handle primitive types, mathematics, program invocation, and supervision of applications.
`System::Collections`	Contains classes that define collections of objects, such as lists, queues, arrays, hash tables, and dictionaries.
`System::Collections::Generic`	Contains classes that allows classes, structures, interfaces, methods, and delegates to be declared and defined without specific types.
`System::Collections::Specialized`	Contains classes that specialize collections of objects, such as string collections, string dictionaries, and name-value collections.
`System::Configuration`	Contains classes that provide access to configuration data found in the `app.config` and `web.config` files.
`System::Data`	Contains classes that handle database access.

Table 1-2. *Common .NET Framework Class Library Namespaces (Continued)*

Namespace	Description
`System::Data::OleDb`	Contains classes that handle access to OLE DB databases.
`System::Data::OracleClient`	Contains classes that handle access to Oracle databases.
`System::Data::SqlClient`	Contains classes that handle access to Microsoft SQL Server databases.
`System::Diagnostics`	Contains classes that allow you to debug your application and trace application execution.
`System::DirectoryServices`	Contains classes to access Active Directory.
`System::Drawing`	Contains classes to handle the GDI+ graphics functionality.
`System::Drawing::Drawing2D`	Contains classes that handle advanced two-dimensional and vector graphics functionality.
`System::Drawing:Imaging`	Contains classes to handle advanced GDI+ imaging functionality.
`System::Drawing::Printing`	Contains classes to handle custom printing.
`System::Globalization`	Contains classes that define culture-related information, such as language, currency, and numbers.
`System::IO`	Contains classes to handle reading and writing of data streams and files.
`System::Media`	Contains classes to play `.wav` sound files or system sounds.
`System::Net`	Contains classes to handle many of the protocols and services found on networks.
`System::Reflection`	Contains classes that examine loaded types, methods, and fields, and also dynamically create and invoke types.
`System::Resources`	Contains classes to create, store, and manage various culture-specific resources.
`System::Runtime::InteropServices`	Contains classes to access COM objects and native APIs.
`System::Runtime::Remoting`	Contains classes to create and configure distributed applications.
`System::Text`	Contains classes representing multiple character encodings used to encode and decode blocks of bytes to and from string. You will also find `StringBuilder` here, which you should use to build strings dynamically.
`System::Security`	Contains classes to handle the CLR security system.
`System::Threading`	Contains classes to handle multithreaded programming.
`System::Timers`	Contains classes to raise an event on a specified interval.

Table 1-2. *Common .NET Framework Class Library Namespaces (Continued)*

Namespace	Description
`System::Web`	Contains classes to handle communication between browser and server.
`System::Web::Mail`	Contains classes to create and send an e-mail using the SMTP mail service built into Microsoft Windows 2000.
`System::Web::Security`	Contains classes to handle ASP.NET security in Web applications.
`System::Web::Services`	Contains classes to build and use Web services.
`System::Web::UI`	Contains classes to create controls and pages in Web applications.
`System::Windows::Forms`	Contains classes to create Windows-based applications.
`System::XML`	Contains classes to handle you've XML.

A Sad Note About C++/CLI Support of 3.0 and 3.5 Application Development Technologies

You may have noticed the lack of any namespaces in Table 1-2 that support Windows Presentation Foundation, Windows Communication Foundation, Windows Workflow Foundation, and LINQ. This is not an omission on my part. The reason is, as the Visual C++ team puts it

> *Moving forward we will increase our support for native development tools and will work to provide "friction-free" Interop between native & managed code.… With this change we will focus less on making Visual C++ a "pure" .NET development tool. This decision is supported by the fact that Microsoft already has tools in Visual C# and Visual Basic that meet the needs of .NET developers. By not having to map to all the new aspects of.NET (such as LINQ or WPF designers), our team can concentrate on building better native and Interop features. Despite these changes, please note that we will continue to provide support for C++/CLI as it is fundamental to our Interop strategy.*

In a nutshell, C++/CLI will not support these newer features directly and instead will focus on Interop. I think this is rather sad and possibly shortsighted. But since I don't work at Microsoft, I don't have any say in the matter. Anyway… since Microsoft isn't planning on supporting these features, directly with C++/CLI, I didn't think it worth my time or yours devoting multiple chapters on them in this book.

On the other hand, as a C++/CLI programmer you are not completely left out in the cold. It is still possible to write all these new and wonderful technologies in C# and then expose them in C++/CLI. Oh, you can also develop WPF Extensible Application Markup Language (*XAML*) in the C# design tool and then use it in C++/CLI. (I'm not sure why you would do this, but hey, the ability is available if you want it.)

Summary

This chapter created a level playing field on which to start your exploration of C++/CLI, beginning with the big picture, examining what exactly .NET is. I then explored the .NET Framework generically and finally broke it down piece by piece, examining such things as assemblies, the common language runtime (CLR), the common type system (CTS), and the common language specification (CLS). The chapter ended with a look at the myriad classes available to the C++/CLI developer.

The journey has begun. In the next chapter, you'll look at the basics of C++/CLI. Let's continue.