CHAPTER 1

Why C++/CLI?

The road to C++/CLI began some 30 years ago with a very small C program:

```
// HelloWorld.cpp
// compile with "CL HelloWorld.cpp"

#include <stdio.h>

int main()
{
  printf("hello, world");
}
```

To be precise, this code is not just a C program, but also a C++ program, since C++ derived from C. Because C++ has a high degree of source code compatibility with C, you can mix many C constructs with C++ constructs, as the following code shows:

```
// HelloWorld2.cpp
// compile with "CL /EHs HelloWorld2.cpp"

#include <stdio.h>
#include <iostream>

int main()
{
  printf("hello");
  std::cout << ", world";
}
```

Extending C++ with .NET Features

In a very similar way, C++/CLI is layered on top of C++. C++/CLI provides a high degree of source code compatibility with C++. As a consequence, the following code is valid if you build the program with a C++/CLI compiler:

```
// HelloWorld3.cpp
```

```
#include <stdio.h>
#include <iostream>

int main()
{
  // use a C function to print "hello"
  printf("hello");

  // use a C++ object  to print a comma followed by a space
  std::cout << ", ";

  // use a .NET class to print "world"
  System::Console::WriteLine("world");
}
```

The interesting aspect of this code is that it uses C, C++, and .NET constructs to implement `main`. First, it calls the native function `printf` to write the string `"hello"`, then it uses the C++ object `std::cout` to write `", "`, and finally it uses the .NET class `System::Console` to write the last part of that well-known message.

What Is .NET?

Before looking at the steps necessary to build the preceding application, I should cover what the term *.NET* means and what it offers to a software developer. .NET is an infrastructure that provides two major benefits: productivity and security. Using .NET, a developer can write code for many modern problem domains faster, and during coding, the developer faces fewer pitfalls that could end up in security vulnerabilities. Furthermore, .NET code can be implemented so that it can be executed with restricted access to APIs. All these benefits are achieved by two components: a runtime and a class library.

The .NET runtime and core parts of the base class library are specified as an open standard. This standard is called the *Common Language Infrastructure (CLI)*, and it is published as the ECMA-335 standard and as the ISO standard 23271. There are several implementations of this standard. The *Common Language Runtime (CLR)* is the most important implementation because it is the most powerful one, and it targets Microsoft Windows operating systems, the most common platform for .NET development.

In the context of .NET, you very often hear the term *managed*. .NET code is often called managed code, .NET types are managed types, objects in .NET are managed objects, and for the heap on which managed objects are instantiated, the term *managed heap* is used. In all these cases, the term *managed* means "controlled by the .NET runtime." The .NET runtime influences most aspects of managed execution. Managed code is JIT-compiled to machine-specific (native) code. For managed types, you can easily retrieve runtime type information, and managed objects are garbage-collected. (Memory management is discussed in Chapter 2, and other runtime services are covered in Chapter 4.)

To differentiate between .NET concepts and non-.NET concepts, the term *unmanaged* is used quite often.

What Is C++/CLI?

C++/CLI is a set of extensions made to the C++ language to benefit from the services that an implementation of the CLI offers. These extensions are published as the ECMA-372 standard. With the help of these extensions, a programmer can use .NET constructs in existing C++ code, as shown previously. Visual C++ 2005 implements the C++/CLI standard to support executing code on the CLR.

Building C++/CLI Applications

To make the switch from C to C++, a new file extension was used. As you can see in the preceding `HelloWorld3.cpp` example, the file extension for C++/CLI applications remains unchanged. However, there is still a need to distinguish between C++ compilation and C++/CLI compilation—the result of native C++ compilation is native code, whereas the result of C++/CLI compilation is managed code. If you try to compile the code on the command line, as shown in the following, you'll get compiler errors.

```
CL.EXE HelloWorld3.cpp
```

These errors will complain that `System` is neither a class nor a namespace name, and that the identifier `WriteLine` is not found. Both the namespace `System` and the method `WriteLine` are the managed aspects of your code. The Visual C++ compiler can act as a normal C++ compiler or as a C++/CLI compiler. By default, it remains a native compiler. To use it as a C++/CLI compiler, you use the compiler switch `/clr`, as in the following command line:

```
CL.EXE /clr HelloWorld3.cpp
```

This simple HelloWorld3 application shows one of the advantages that C++/CLI has over all other commonly used .NET languages: it provides source code compatibility with a good old native language.

C++/CLI is a superset of the C++ language. A valid C++ program is also a valid C++/CLI program. As a consequence, your existing code base is not lost. Instead of reimplementing existing applications with a completely new language and programming infrastructure, you can seamlessly extend existing code with .NET features.

The `HelloWorld3.exe` file created by the C++/CLI compiler and linker is a so-called .NET assembly. For this chapter, it is sufficient to consider assemblies as the deployable units of the .NET world. Chapter 4 will provide a more detailed definition of this term. The `HelloWorld3.exe` assembly differs from assemblies created by other .NET languages because it contains native code as well as managed code. An assembly like `HelloWorld3.exe` is also called a *mixed-code assembly*.

A migration strategy based on C++/CLI can preserve huge investments in existing C++ source code. This is extremely important because there is a vast amount of C++ code that is already written, tested, accepted, and in service. Furthermore, this strategy allows a partial migration with small iterations. Instead of switching everything to .NET in one chunk, you can flexibly use different .NET features when they seem appropriate.

Object File Compatibility

Partial migration obviously depends heavily on source code compatibility. Once existing C++ source code is compiled to managed code, you can straightforwardly and seamlessly integrate other .NET components and benefit from the many features the .NET Framework offers. However, there is a second pillar that you must understand to use C++/CLI efficiently. I like to refer to this feature as *object file compatibility*.

Like source code compatibility, the object file compatibility feature of C++/CLI has an interesting analogy to the shift from C to C++. As shown in Figure 1-1, the linker accepts object files compiled from C and C++ sources to produce a single output.

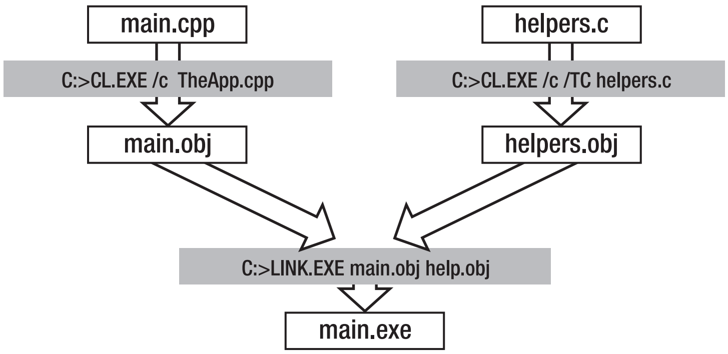


Figure 1-1. *Linking C and C++ based object files into an application*

The compiler switch `/c` produces just an object file instead of an executable output. In this sample, one file is compiled with the C++ compiler and a second file is compiled with the C compiler. (The `/TC` switch is used for that.) Both resulting object files are linked to produce the application.

When a source file is compiled with `/clr`, the compiler produces a managed object file. Equivalent to the scenario with C and C++ based object files, the linker can get managed and unmanaged object files as an input. When the linker detects that at least one input is a managed input, it generates a managed output. Figure 1-2 shows how you can link a managed and an unmanaged object file into a single output file.

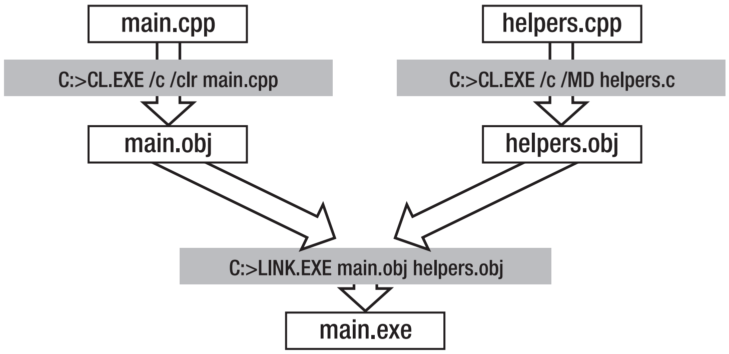

main.cpp
C:>CL.EXE /c /clr main.cpp
main.obj
helpers.cpp
C:>CL.EXE /c /MD helpers.c
helpers.obj
C:>LINK.EXE main.obj helpers.obj
main.exe

Figure 1-2. *Linking managed and unmanaged object files into an application*

As Figure 1-3 demonstrates, you can also create a managed static library that itself can be an input to the linker. In this figure, `TheLib.obj` is a managed object file. Therefore, `TheLib.lib` is a managed static library. `TheApp.obj` is also a managed object file. The linker gets two managed inputs, so the resulting `TheApp.exe` is a .NET assembly. Feel free to ignore the keyword `__clrcall` in `TheApp.cpp`. It will be discussed extensively in Chapter 9.

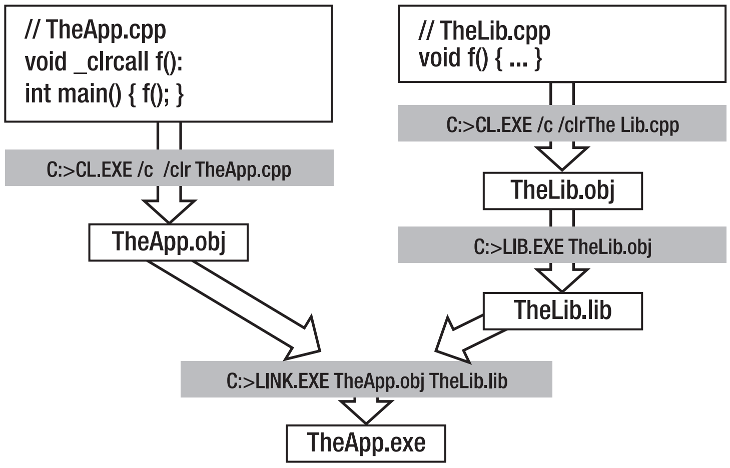


Figure 1-3. *Linking with managed library inputs*

Object file compatibility is an important feature because it allows you minimize the overhead of managed execution. If you use C++/CLI to integrate .NET code into existing C++ code, you will likely decide to compile most or even all of your existing code without `/clr`, and to compile only new files to managed code. In Chapter 7, I will explain what you should consider before you modify your project settings to turn on the `/clr` compiler switch, and I will give step-by-step instructions for modifying your project configurations.

Interaction Between Managed and Unmanaged Code

Linking native code and managed code into one assembly is only useful if native code can call managed code and vice versa. Here is an example that shows how easy this is. Assume you have a file like this one:

```
// UnmanagedCode.cpp
// compile with "CL /c /EHs /MD UnmanagedCode.cpp"

#include <iostream>
using namespace std;

void fUnmanaged()
{
  cout << "Hello again from unmanaged code.\n" << endl;
}
```

If you compile this file with the command mentioned in the comment, you will get an unmanaged object file named `UnmanagedCode.obj`. Obviously, `fUnmanaged` will be compiled to unmanaged code. Although `fUnmanaged` is not a managed function, you can seamlessly call it in a file compiled to managed code. The only thing you need is the function declaration for `fUnmanaged`. Under the hood, the C++/CLI compiler and the CLR do several things to make this possible, but at the source code level, there is nothing special to do. The next block of code shows a managed source file that calls `fUnmanaged`:

```
// ManagedCode.cpp
// compile with "cl /c /clr ManagedCode.cpp"

extern void fUnmanaged();      // implemented in UnmanagedCode.cpp

void fManaged()
{
  System::Console::WriteLine("Greetings from managed code!");
  fUnmanaged();
}
```

The next example shows a native caller for the managed function `fManaged`. Again, only the normal function declaration is necessary, and under the hood the compiler and the runtime make sure the code works as intended.

```
// HelloWorld4.cpp
// compile with "cl /MD HelloWorld4.cpp /link ManagedCode.obj UnmanagedCode.obj"

#include <stdio.h>

extern void fManaged();    // implemented in ManagedCode.cpp

int main()
{
  printf("Hi from native code.\n");
  fManaged();
}
```

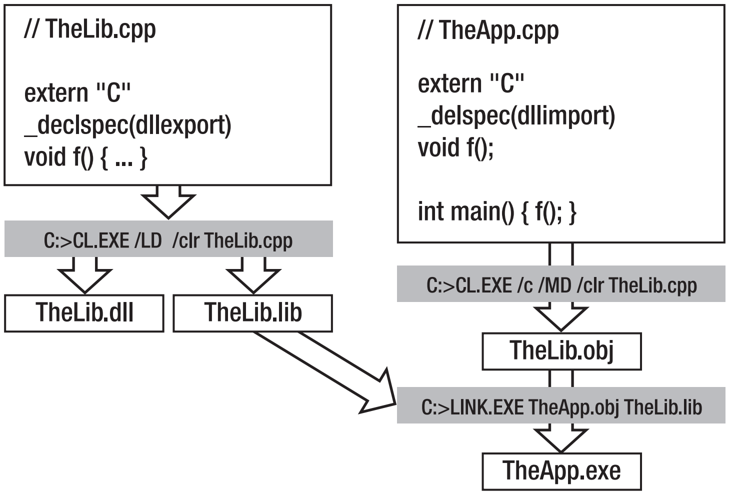
If you use the command shown in the comment of this code, the C++/CLI compiler will generate a native object file, `HelloWorld4.obj`, and link it together with `ManagedCode.obj` and `UnmanagedCode.obj` into the application `HelloWorld4.exe`. Since `HelloWorld4.obj` is a native object file, `HelloWorld4.exe` has a native entry point. The `printf` call in `main` is a native call from a native function. This call is done without a switch between managed and unmanaged code. After calling `printf`, the managed function `fManaged` is called. When an unmanaged function like `main` calls a managed function like `fManaged`, an unmanaged-to-managed transition takes place. When `fManaged` executes, it uses the managed `Console` class to do its output, and then it calls the native function `fNative`. In this case, a managed-to-unmanaged transition occurs.

The HelloWorld4 application was written to explain both kinds of transitions. It is extremely helpful to have both these options and to control in a very fine-grained way when a transition from managed code to unmanaged code, or from unmanaged code to managed code, takes place. In real-world applications, it is important to avoid these transitions, because method calls with these transitions are slower. Reducing these transitions is key to avoiding performance penalties in C++/CLI. To detect performance problems, it can be very important to identify transitions between managed and unmanaged code. Reducing performance penalties is often done by introducing new functions that replace a high number of managed/unmanaged transitions with just one. Chapter 9 gives you detailed information about internals of managed/unmanaged transitions.

The code samples used so far have used the simplest possible method signature. However, the interoperability features of C++/CLI allow you to use any native type in code that is compiled to managed code. This implies that methods called via managed/unmanaged transitions can use any kind of native type. Chapter 8 will discuss all details of type transitions.

DLLs with Managed Entry Points

You can also factor managed code out into a separate DLL so that your existing projects remain completely unmanaged. Figure 1-4 shows a simple example of such a scenario.



```
// TheLib.cpp

extern "C"
_declspec(dllexport)
void f() { ... }
```

```
C:>CL.EXE /LD  /clr TheLib.cpp
```

```
// TheApp.cpp

extern "C"
_delspec(dllimport)
void f();

int main() { f(); }
```

```
C:>CL.EXE /c /MD /clr TheLib.cpp
```

```
C:>LINK.EXE TheApp.obj TheLib.lib
```

Figure 1-4. *Separating managed code in DLLs*

In this simple scenario, `TheApp.cpp` shall represent your existing project. To extend it with managed features, a new DLL is created from the source file `TheLib.cpp`. Notice that `TheLib.cpp` is compiled with `/clr`. Therefore, the exported function `f()` is a managed function. When `main` calls the managed function `f`, the CLR is delay-loaded.

Using mixed-code DLLs like `TheLib.dll` from the preceding sample, you can minimize the impact that managed execution has on your project. However, there are some pitfalls that you should know before you start writing mixed-code DLLs. Chapter 12 gives you all the necessary information to avoid these pitfalls.

Compilation Models

So far, I have discussed the following two major features, sometimes summarized as C++/CLI interoperability:

- Existing C++ source code can be compiled to managed code (source code compatibility).
- Native code and managed code can be linked into a mixed-code assembly (object file compatibility).

Compared to the interoperability features that other .NET languages provide, C++/CLI interoperability is much more powerful. It is a significant simplification for interoperating with native code, and it enables developers to save huge investments of existing C++ code.

On the other hand, these powerful features have side effects. Often, these side effects can be ignored, but it is also possible that these side effects are incompatible with other constraints and requirements of a project. To handle situations that are incompatible with the side effects caused by C++/CLI interoperability, Visual C++ allows you to turn either the object file compatibility or both C++/CLI interoperability features off. This can be done by choosing different compilation models supported by Visual C++. If the command-line option `/clr` is not used, the compiler chooses the native compilation model. To compile to managed code, the compiler argument `/clr`, or one of its alternatives—`/clr:pure` or `/clr:safe`—can be chosen.

As shown in the preceding samples, the `/clr` compiler option enables you to use both interoperability features mentioned previously. The compiler option `/clr:pure` still allows you to compile existing C++ code to managed code (source code compatibility), but you cannot produce mixed-code assemblies, which would require object file compatibility. The linker does not allow you to link object files produced with `/clr:pure` with native object files. An assembly linked from object files compiled with `/clr:pure` will have only managed code; hence the name.

Assemblies containing only managed code can be used to bypass two special restrictions of mixed-code assemblies. However, these restrictions apply only to very special scenarios, and understanding them requires knowledge of .NET features discussed later in this book. Therefore, I defer this discussion to Chapter 7.

Another restriction that applies to mixed-code assemblies as well as to assemblies built with `/clr:pure` is much more relevant: neither kind of assembly contains verifiable code, which is a requirement for .NET's new security model, called Code Access Security (CAS). CAS can be used to execute assemblies with restricted abilities to use features of the runtime and base class libraries. For example, pluggable applications are often implemented so that plug-ins do not have any permission on the file system or the network. This is sometimes called *sandboxed execution*.

Certain features of the runtime could be misused to easily bypass a sandbox of restricted permissions. As an example, all features that allow you to modify random virtual memory could be used to overwrite existing code with code that is outside of the runtime's control. To ensure that none of these dangerous features are used by a sandboxed assembly, its code is verified before it is actually executed. Only if code has passed the verification can it be executed in a sandbox. The powerful interoperability features that are supported with the compilation models `/clr` and `/clr:pure` use nonverifiable features intensively. To produce verifiable code, it is required to use the compilation model `/clr:safe`. Source code that is compiled with `/clr:safe` can only contain .NET constructs. This implies that native C++ types cannot be used.

Wrapping Native Libraries

C++/CLI is not only the tool of choice for extending existing C++ applications with .NET features, but it is also the primary tool for creating mixed-code libraries. These libraries can be simple one-to-one wrappers for native APIs; however, in many scenarios it is useful to do more than that. Making existing C++ libraries available to .NET developers so that they can make the best use of them requires giving an existing API a .NET-like face.

In .NET, there is a new type system with new features, and there are also new philosophies related to class libraries, error reporting, data communication, and security that all have to be considered to make a wrapper a successful .NET library. Several chapters of this book are dedicated to different tasks of wrapping native libraries. Chapters 5 and 6 explain how to define the various kinds of managed types and type members, and show the .NET way to map different kinds of relationships to a system of managed types. In Chapter 10, important design and implementation aspects of wrapper libraries are discussed. Finally, Chapter 11 explains how to use the reliability features provided by .NET to ensure that the wrapped resources are cleaned up even in critical scenarios like stack overflows. High reliability is of special importance if a wrapper library is used in long-running servers.

Summary

This chapter has introduced some of the salient features of C++/CLI. Using C++/CLI, you can combine the good old world of native C++ and the fancy new world of .NET. As you've seen, you can integrate managed code into existing C++ code and also link native and managed inputs into a single output file. Furthermore, you can seamlessly call between managed and unmanaged code. These features are extremely useful, both for extending existing applications and libraries with .NET features and for writing new .NET applications and libraries that require a lot of interoperability between managed and unmanaged code.