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

Copyright © 2008 by Gordon Hogenson

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

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

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

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

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

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

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

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

CHAPTER 1

Introducing C++/CLI

This chapter introduces the C++/CLI language extensions to C++ and shows you the classic "Hello, World" example in C++/CLI. You'll learn just enough about the runtime environment that executes your C++/CLI programs to get started with your first program. You'll also learn some of the features available in that environment, including access to the .NET Framework (or the CLI class libraries), the common type system, and other helpful features such as garbage collection.

Garbage Collection and Handles

One convenience of a managed language is *garbage collection*—that you no longer have to keep track of all the objects you create. Your C++/CLI objects will be collected and destroyed by a background process called the *garbage collector*. Think about this analogy for a minute. When civilization in an area reaches a certain point, your household waste is collected conveniently at the curbside for burial, incineration, or recycling. As important as garbage collection is, the implications or benefits of the common language runtime (CLR) don't stop at garbage collection. In this analogy, a civilized environment has other implications as well. There is a government to contend with, which has its benefits and drawbacks. Taxes might be higher, but you get all kinds of services such as telephones, electricity, and a reliable water supply. Similarly, for your program, you might pay a performance penalty; however, you get a lot in return in terms of functionality that makes life easier as a programmer.

Remember that C++/CLI, unlike other languages that also target the CLR, doesn't replace standard C++ functionality. C++/CLI not only adds the ability to create managed objects, but also allows the creation of C++ objects, called *native objects*. But since both entities exist in the language, how are you to distinguish them? The answer is that instead of using pointers, you use *tracking handles*. Tracking handles are very similar to pointers, but they behave differently since they refer to managed objects, not native objects.

There are two entirely separate families of types in C++/CLI—the *native type system* exists fully intact alongside the *managed type system*. Objects or instances of native types can coexist in the same application with objects and instances of managed types. Whether a type is native or managed depends on whether it is declared with C++ syntax or with the C++/CLI syntax for managed types. Chapter 2 covers this in detail, but just to get started, instead of `class`, `ref class` is used for a managed reference class.

```
class N { ... };
ref class R { ... };
N* n = new N;    // standard C++ pointer to an object
R^ r = gcnew R;  // C++/CLI handle to an object
```

Recall that native objects, when created with the `new` statement (or `malloc`), are allocated on a large pool of memory called the *heap*. It's important to understand that there are actually two heaps in a C++/CLI application, the native heap and the managed heap. The *native heap* is used when you use the `new` statement, as usual, to create instances of your native classes. As in standard C++, you must explicitly manage the lifetime of the objects on this heap yourself. The *managed heap* is a separate pool of memory that is managed by the garbage collector. Instead of a normal pointer into the native heap, you use a tracking handle to point to objects in the managed heap. A tracking handle is expressed using the caret symbol (^), instead of the asterisk (*). Also, instead of `new`, the keyword `gcnew` is used. As you might have guessed, the "gc" stands for "garbage collected."

The reason these new pointer-like entities are called tracking handles is that in addition to freeing up unusable objects, the garbage collector also moves objects around in memory in order to organize the heap so that its operations can be carried out more efficiently. This is called *heap compaction*. This means that, unlike a native pointer, the address pointed to by a tracking handle, and therefore the location of the object it tracks, may change in the course of the program. For this reason, you don't normally access the address of an object pointed to by a tracking handle. The runtime will update the address of any tracking handles if the garbage collector moves your object. From this point on, for brevity, I'll refer to them simply as *handles*.

There are certainly many parallels between pointers and handles; however, you must not assume that a handle is simply a "managed pointer." There are some subtle differences between the two, as you'll see in Chapter 4.

In general, the managed, garbage-collected environment makes for less detailed memory management work for developers, who no longer have to worry about making sure they free all allocated memory. Some might assert that this makes programmers lazy. I recall that in Plato's dialogue *Critias*, the same argument arose among the ancient Egyptians over the Egyptian god Thoth's gift to mankind, the gift of writing. Some scholars at the time said that this was surely the end of memory, for the crutch of the written word would surely replace the need for memorization. All I can say is that some people's response to progress hasn't changed much in 6,000 years.

I'll refer to the C++ features that predate the C++/CLI extensions as *classic C++*. I'll use the word "managed" to describe anything governed by the CLR (or another implementation of the CLI): managed code, managed types, managed pointers, and so on. However, the term *managed C++* should not be used to describe the new language syntax. With a few exceptions, every feature of classic C++ is also a feature of C++/CLI, so it's not true to say that C++/CLI is only a managed language. The word "native" refers to the unmanaged world, hence I use the terms *native types*, *native compilation*, and so on. The term *native C++* could be used to refer to the C++ language without the extensions, but since the new language supports both managed and native types, I prefer the term *classic C++*.

The /clr Compiler Option

If you use Visual C++ 2005 or 2008, you have to let the compiler know that you are targeting the CLR (and therefore want C++/CLI standard extensions enabled). You do this by using the `/clr` compiler option (or one of its variants, as discussed in Chapter 3). In the Visual C++ 2005 or 2008 development environment, you would choose the appropriate type of project, and the option would be set appropriately for that project type. If you need to change the option later, you can set the Common Language Runtime support option in the General tab of the Project Properties dialog.

The Virtual Machine

C++/CLI applications execute in the context of the CLR. The CLR implements a *virtual machine*, which is a software implementation of an idealized, abstract execution environment. Programs that run on the CLR virtual machine use a language known as the Common Intermediate Language (CIL). Microsoft's implementation of CIL is often referred to as MSIL, or just plain IL. The CLR relies on a JIT (just-in-time) compiler to translate the CIL code on demand into machine code in order to execute the program.

The CLR virtual machine is Microsoft's implementation of the *Virtual Execution System* (VES), which is part of the ECMA standard. As processors change, you need only change the way in which the executable code is generated from the processor-independent layer, and you'll still be able to run the old programs written for the earlier processor. Pure IL generated by compilers targeting the CLR does not contain x86 instructions or any other object code that is configured to run on a particular processor. Compilers output MSIL code that can run on the virtual machine.

You'll see in Chapter 3 that there are three compilation modes available: *mixed mode*, *pure mode*, and *safe mode*. Each of these modes differs in the native code that is allowed, and each has advantages and disadvantages. Later you'll learn in more detail when to use each option. For now, remember that there are many degrees of managed code. It is often assumed that once you transition to the CLR, all the problems (and freedoms) of the native code world are left behind. That is not true—you can run almost all classic C++ source code on the virtual machine just by recompiling it with the `/clr` option. The only difference is that your code is compiled to IL instead of the assembler in between. Ultimately, it all boils down to machine code being executed by the processor.

The real benefits of the managed world come not with recompiling your existing classic C++ code, but by using the C++/CLI constructs that constitute a system of object types uniquely suited to do well in the managed world.

The Common Type System

The CLR type system is mirrored in C++/CLI, so it's important to understand how it works.

The CLR has a unified type system called the *common type system* (CTS). A unified type system has at its root a single type, often called `Object`, from which all types are derived. This is very different from the C++ type system, sometimes called a *forest*, in which there may be arbitrarily many independent type hierarchies.

The CTS represents a set of type relationships that many C++ programmers will find unfamiliar. There is no multiple inheritance, and only reference classes can be allocated on the managed heap and support inheritance and virtual functions. Chapter 2 will explain these differences. I'll use the term managed type to mean any type that is part of the CLR's type system. The C++/CLI type system for managed types is designed to allow the use of managed types in C++/CLI programs. Because all managed types must inherit (directly or indirectly) from the root type, `Object`, even the primitive types used in managed code (the managed versions of `int`, `double`, etc.) have a place in this type system, in the form of objects that wrap or "box" each primitive data type. The base class library defines a namespace called `System`, which contains fundamental types and other commonly used constructs. In fact, the CTS defines most primitive types in the `System` namespace with their own names, such as `System::Int32`. These types are common to all languages using the CLR. The primitive C++/CLI types such as `int` are synonyms for the equivalent .NET Framework types (e.g., `int` is synonymous with `Int32`), so that you have the convenience of referring to the type using the same name you'd use in C++. You can use two ways to refer to most primitive types. In Chapter 2, you'll learn how the primitive types in C++ map to the CLI common type system.

Reference Types and Value Types

Every managed type falls into one of two categories: *reference types* or *value types*. The difference between value types and reference types is that value types have value semantics while reference types have reference semantics. Value semantics means that when an object is assigned (or passed as a parameter), it is copied byte for byte. *Reference semantics* means that when an object is assigned (or passed as a parameter), the object is not copied; instead, another reference to that same object is created.

Value types are used for objects that represent a value, like a primitive type or a simple aggregate (e.g., a small structure), especially one that is to be used in mathematical computations. Computations with value types are more efficient than with reference types because reference types incur an extra level of indirection; reference types exist on the heap and can only be accessed through the handle, while the value type holds its value directly. Value types actually live in a limited scope, either as a local variable at function scope or in the scope of another object as a field. They also do not have the overhead of an object header, as reference types do. However, value types are limited in many ways. Value types are often copied—for example, when used as a method parameter, a copy is automatically created—so they are not suitable for large objects; they also cannot be used in inheritance hierarchies, and they don't support more complex and powerful object operations such as copy constructors, nontrivial default constructors, assignment operators, and so on.

Reference types are used wherever reference semantics are required and when modeling more complex objects for which the limitations of value types are too restrictive. They may inherit from another class and may in turn be inherited from. They are not copied byte for byte (for example, when passed as an argument to a function), rather, they are passed as references, so they may be large and not suffer a penalty from excessive copying. They can have special member functions such as default constructors, destructors, copy constructors, and the copy assignment operator (although neither type can have overloaded operators `new` and `delete`). The actual objects live on the managed heap. The handle itself is just an address that refers to the object's header (which is 8 bytes in size for the 32-bit CLR) on the heap.

Figure 1-1 shows the memory layout of a typical value type and a reference type.

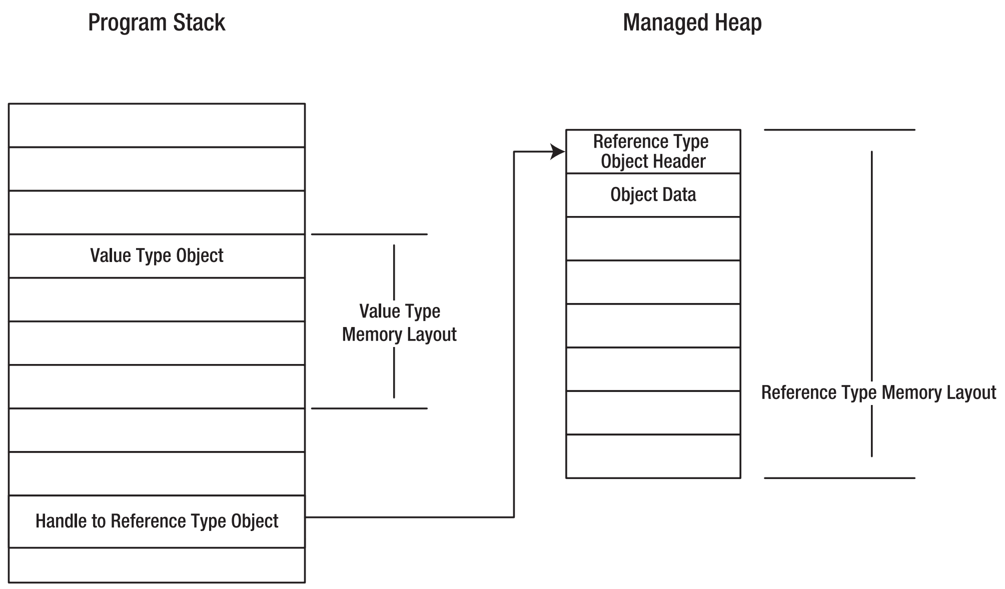
Program Stack
Managed Heap
Value Type Object
Value Type Memory Layout
Handle to Reference Type Object
Reference Type Object Header
Object Data
Reference Type Memory Layout

Figure 1-1. *Storage characteristics of reference types and value types. Value types are shown here on the stack (although they could also be a part of an object on the managed heap). Reference types involve a handle plus an object on the managed heap.*

The CLI and the .NET Framework

The CLI includes the VES and a standardized class library, often called the *base class library* (BCL), that provides support for fundamental programming. The .NET Framework is a large class library released by Microsoft that implements the base class library as well as additional functionality that isn't part of the ECMA standard. If you are using Visual Studio and targeting the CLR, you have access to the .NET Framework class libraries within your C++/CLI code. If you are using a different implementation of C++/CLI than Microsoft's, you still have the base class library. This book will not attempt to cover all that the .NET Framework, or even the base class library, allows you to do; however, it will cover basic input and output (Chapter 5), the collection classes (Chapter 11), some of the exceptions, some of the metadata that can be applied to types, and ways of getting information on types at runtime (reflection), all in Chapter 10, as well as other useful aspects of the .NET Framework as necessary.

The full .NET Framework contains support for database access, XML, web services, web pages, Windows application development, and so on. At the time of writing, the newest version of the .NET Framework is version 3.5, which has support for a UI library called *Windows Presentation Foundation* (WPF), used for building Windows applications, and a communications library based on web services called Windows Communication Foundation (WCF). These new features don't affect the C++/CLI developer significantly, since these features are best used from C# or Visual Basic .NET, where full designer support exists for creating WPF applications, for example.

Likely of more interest for C++/CLI developers are new features that ship with Visual Studio 2008, which corresponds to .NET Framework 3.5, such as the marshaling library for simplifying converting data between managed types and native types, and STL/CLR, making the full power of the Standard Template Library available to C++/CLI programs.

"Hello, World"

Now let's look at our first program (Listing 1-1) and see how the language looks in actual code.

Listing 1-1. *"Hello, World"*

```
// hello_world1.cpp
int main()
{
   System::Console::WriteLine("Hello, World!");
}
```

The program in Listing 1-1 illustrates the classic "Hello, World" application. It shows several features from classic C++—a method call with a string argument, the qualification of a method name by the class and the namespace to which it belongs (with the usual double-colon scope operator), and the `main` method. It shows a few features new to the .NET Framework, such as the `System` namespace, the `Console` class, and the `Console` class's `WriteLine` method. You'll notice that there is no `#include` directive. Instead, managed type libraries in C++/CLI are referenced from their compiled form with `#using`.

You could also write this program as shown in Listing 1-2.

Listing 1-2. *"Hello, World" with #using Directive*

```
// hello_world2.cpp
#using "mscorlib.dll"
using namespace System;

int main()
{
   Console::WriteLine("Hello World!");
}
```

The `#using` directive references the DLL file `mscorlib.dll`. The program also employs the `using` declaration in the classic C++ sense, which as you know is simply used to avoid having to use fully qualified names for program elements in the `System` namespace. The `#using` directive is a new C++/CLI concept used to reference the types contained in a DLL. This is very different from `#include`, which references types declared before compilation. The first example you saw works because the compiler automatically inserts `#using "mscorlib.dll"`. This is convenient since nearly all CLI programs require the types that it defines. The DLL is a CLI assembly, which contains not just executable code, but also metadata that exposes information about the types and program elements in the assembly. No header file is needed.

Listing 1-3 illustrates a few more features of the language.

Listing 1-3. *More C++/CLI Features*

```
// hello_world3.cpp

using namespace System;
```

```
ref class Hello
{
    String^ greeting;

    public:

    Hello(String^ str) : greeting(str)
    { }

    void Greet()
    {
        Console::WriteLine(greeting + "!");
    }

    void SetGreeting(String^ newGreeting)
    {
       greeting = newGreeting;
    }
};

int main()
{
   Hello^ hello = gcnew Hello("Hi there!");
   hello->SetGreeting("Hello World");
   hello->Greet();
   hello->SetGreeting("Howdy");
   hello->Greet();
}
```

This code creates a reference class, as indicated by the `ref` keyword. It's called `Hello`, with a constructor, a method called `Greet`, and another method called `SetGreeting`. The `SetGreeting` method takes a `System::String` parameter. The caret indicates that the parameter type is "handle to `String`." The `String` class is the CLI version of a (Unicode) character string. Unlike a native string, the `String` object is invariant, which means it cannot be changed without creating a brand new string. In Chapter 5, you'll see how to create a string that can be manipulated and changed.

■ **Note** Actually, `ref` is not a keyword in exactly the same sense as a C++ keyword. For one thing, it is sensitive to the context in which it is used. Unlike keywords, context-sensitive keywords introduced in C++/CLI can be used as variable names without causing program errors. Also, keywords like `ref class` are considered *whitespace keywords*, which obey certain special rules. See the appendix for information about context-sensitive keywords and whitespace keywords.

Also notice the `Greet` method uses a new C++/CLI method of concatenating strings using the + operator. Also, the constructor and the `SetGreeting` method take a `String`, but the code passes a string literal. The compiler creates a `String` object from the string literal passed in. You'll learn the details of how this works in Chapter 5, but for now just notice that you can use string literals in a natural way with the `String` type, without concerning yourself with the subtleties of whether it's a narrow or wide character string literal.

Just as in classic C++, the `main` method does not need to explicitly return a value, even though its return value is properly `int`, not `void`. This is because the compiler inserts `return 0;` automatically.

In the main method in Listing 1-3, you saw a very important pattern that is used throughout all C++/CLI code. The `Hello` class is a reference type, lives on the managed heap, is created with `gcnew` instead of new, and is referred to using a handle, a named object that refers to the unnamed object on the managed heap. The indirection operator is used, just as if the handle were a pointer to the object. Notice that there is no call to any form of `delete` either.

I've demonstrated a simple reference type, but you may be wondering whether the `Hello` class could also be a value type. Indeed, it can be, because it has no explicit inheritance relationship with any other class (although, because it is a managed type, it implicitly inherits from `Object`); it has no special initialization that would require you to define a special default constructor; it has no other special member functions; and it contains no data. Listing 1-4 shows how the code would look with `Hello` as a value type.

Listing 1-4. *Using a Value Type*

```
// hello_world4.cpp
using namespace System;

value class Hello
{
   // This code is unchanged.
};

int main()
{
   Hello hello;
   hello.Greet("Hello World");
}
```

In the second version, `hello` is created as a local stack variable in the `main` function, rather than on the managed heap, which might result in some performance gain, although with only one object, this hardly matters. Also, a real value type would probably have member variables, perhaps as in Listing 1-5.

Listing 1-5. *A Value Type with Members*

```
value struct Greeting
{
   String^ greeting;
   Char punctuator;
   void PrintGreeting(String^ name)
   {
       Console::WriteLine(greeting + name + punctuator);
   }
};
```

As you can see, the code uses `value struct` in place of `value class`. Throughout this text, whenever I use the term *class*, I mean "class or structure." As in classic C++, one difference between a structure and a class is that structure members are public by default, and class members are private by default.

As you know, the `main` function, also known as the entry point, may take additional arguments that are passed in by the operating system: the number of arguments (traditionally called `argc`) and an array of the arguments as character arrays (traditionally called `argv`). This information is also available to C++/CLI programmers, but instead of using the traditional arguments, you use a managed array type. In this case, the array parameter is an array of handles to `String`, each string representing one of the supplied arguments. The managed array type is one of the many fundamental types defined by the CLR that has special language support in C++/CLI. These CLR analogs of C++ types provide bounds checking, but also are objects in and of themselves, and so provide features called *properties* (discussed in the next chapter), such as the `Length` property used in Listing 1-6, and useful methods. The old `int` parameter of classic C++'s `main` function, `argc`, isn't necessary, since the `Length` property can be used to get the count of command-line arguments.

With this array of arguments, you can supply a person's name on the command line and print a greeting customized to that person, as demonstrated in Listing 1-6.

Listing 1-6. *Using Command-Line Arguments*

```
// greeting.cpp
using namespace System;

value struct Greeting
{
   String^ greeting;
   Char punctuator;

   void PrintGreeting(String^ name)
   {
       Console::WriteLine(greeting + name + punctuator);
   }
};

int main(array<String^>^ args)
{
   Greeting greet;
   greet.greeting = "Hi ";
   greet.punctuator = '!';
```

```
    if (args->Length < 1)
    {
       Console::WriteLine("Enter names on the command line, like this:"
                          " greeting <name1> <name2> ...");
       Console::WriteLine("Use quotes around names with spaces.");
       return 1;
    }

    for (int i = 0; i < args->Length; i++)
    {
       greet.PrintGreeting(args[i]);
    }

    greet.greeting = "Hello, ";
    greet.punctuator = '.';

    for each (String^ s in args)
    {
       greet.PrintGreeting(s);
    }
    return 0;
}
```

As you can see, the type of the array elements is enclosed in angle brackets, and in this case it's a handle to `String`. Why a handle? Because `String` is a reference type. OK, but why are there two handle symbols? The array type is also a reference type, so the outer caret symbol indicates that the argument is a handle to an array.

Listing 1-6 also uses the `for each` statement. The `for each` statement is the semantic equivalent of the `for` loop above it. By eliminating the code for counting, bounds checking, and incrementing, the `for each` statement simplifies performing iteration of an array or other enumerable data structure. In Chapter 9, you'll see how to create data structures that allow the use of `for each`.

Also, notice that the program name is not part of the array, as it is in classic C++. The program name is consumed by the CLR, so it is not available to programs through the parameters to `main`.

Native and Managed Types in the Same Program

Earlier, I said that there was a native type system and a managed type system in C++/CLI. Both families of types can be used in the same program side by side. You won't see this in C#.

Listing 1-7 shows both type systems in action. First, a native C++ class, `HelloNative`, is declared and defined. Next, a managed class, `HelloManaged`, is declared and defined. Both are then used in the `main` function.

Listing 1-7. *Native and Managed Types in the Same Program*

```
// hello_interop.cpp

#include <stdio.h>
#include <string>

class HelloNative
{
   private:
      // string is a Standard C++ class (actually a typedef for basic_string<char>)
      // where each character is represented by a char
     std::string* s;

   public:
     HelloNative()
     {
        s = new std::string("Hello from native type!");
     }
     void Greeting()
     {
        // The C++ basic_string class contains a method that returns a
        // "C string" of type "const char *" for the C runtime function printf.
        printf("%s\n", s->c_str());
     }
     ~HelloNative()
     {
        delete s;
     }
};

ref class HelloManaged
{
   private:
     System::String^ s;

   public:
     HelloManaged()
     {
        s = "Hello from managed type!";
     }
     void Greeting()
     {
        System::Console::WriteLine("{0}", s);
     }
};
```

```
int main()
{
   HelloNative* helloNative = new HelloNative();
   HelloManaged^ helloManaged = gcnew HelloManaged();

   helloNative->Greeting();
   helloManaged->Greeting();
}
```

From the preceding code, you can see that native types can be used freely in C++/CLI programs in the same way as you would use them in classic C++ programs.

Summary

This chapter touched upon the basics of garbage collection and handles. You were introduced to a few terms, learned about the common type system, and saw a simple first program in C++/CLI. You looked closely at the new `main` function, with a brief preview of the managed array and the `for each` statement. You also saw how managed and native types can coexist in a single program.

In the next chapter, you'll explore many more C++/CLI language features in a broad overview.