CHAPTER 6

Data Types

The game's afoot.

—William Shakespeare, Henry V

In this chapter, we dig deeper into the CLI type system as well as the implementation of C++/CLI in Microsoft Visual C++ 2013. You should already have a firm base in C# classes, structures, and interfaces from your previous programming experience, and we want to build on that knowledge to expose differences between C# and C++/CLI.

C# Types versus C++ Types

The C# language was designed specifically to target the CLR. Therefore, all of its data types map directly to CLI types. C++/CLI not only defines data types for the CLI but also defines data types for use in native C++. All of these data types are defined using the keywords `class` and `struct`. These keywords had a preexisting meaning in C++ before C# was devised. In C++, `class` defines a native type, which can be thought of as a general collection of related fields and methods. A C++ `struct` is the same as a C++ `class` except that the default accessibility is `public` for all members, and a C++ `struct` inherits publicly from its base classes.

In C#, `class` defines a CLI reference type descended from `System::Object` that has a specific set of characteristics and limitations. A C# `struct` defines a CLI value type that has a different set of characteristics.

For C++/CLI to implement the CLI types, new keyword combinations were added to the language. The class qualifiers `ref` and `value` are prepended to the keywords `class` and `struct` to create the new whitespace keywords `ref class` and `value class`. These indicate managed types as defined by the CLI. Table 6-1 illustrates the correspondences.[1]

[1]A future version of C++/CLI may implement mixed types; a *mixed type*, according the C++/CLI specification, is "either a native class or a ref class that requires object members, either by declaration or by inheritance, to be allocated on both the CLI heap and the native heap."

Table 6-1. *C++, C#, and CLI Type Comparision*

C++ Type	C# Type	CLI Type	Default Accessibility	Stored In
`ref class`	`class`	reference	`private`	managed heap, stack
`ref struct`	N/A	reference	`public`	managed heap, stack
`value class`	`struct`	value	`private`	native heap, managed heap, stack
`value struct`	N/A	value	`public`	native heap, managed heap, stack
`class`	N/A	N/A	`private`	native heap, stack
`struct`	N/A	N/A	`public`	native heap, stack

The C++ struct Keyword

Let me reiterate that the `struct` keyword is not used in C++/CLI to indicate either a C# `class` or a C# `struct`. A `struct` in C++ is the exact same thing as a `class`, except that it has `public`, rather than `private`, accessibility and inherits publicly by default. This is also true of `ref struct` and `value struct`. Other than with respect to protection mechanisms, they are identical to `ref class` and `value class`.

A C++ `struct` is extremely useful. Whenever I wish to prototype a quick class or method and don't want to worry about protection, I use a C++ `struct` rather than a C++ `class`. I'm going to delve deeper into protection mechanisms in Chapter 8.

Native Classes

As mentioned previously, C++/CLI also has native classes. Unlike a `ref class` and a `value class`, which map directly to CLI defined types, a *native class* is an unmanaged type that does not have a CLI mapping. Native classes, elements of classic C++, will be discussed further in Chapter 18.

Value Types and Reference Types

Value types and reference types are allocated, accessed, and copied with the same syntax; declaring these types as either `struct` or `class` is the main difference. As shown previously, having the same syntax can lead to unintended consequences when instances of these types are assigned. Value types are copied in their entirety, whereas only handles to the reference type are actually copied during an assignment. In C#, except for the initial declaration where you declare `struct` or `class`, the syntax is identical for value types and references types. C# hides the distinction between value types and reference types from the programmer, which can be both good and bad. C++/CLI, on the other hand, does not hide these details and distinguishes between value types and reference types. A quick review of memory allocation during program execution is a great place to start understanding the different concepts.

Dynamic Memory Pools

During the execution of a C# program, new items are allocated in one of two places: the stack or the managed heap. C++/CLI adds a third pool, the native heap, which will be discussed further in Chapter 18 during the discussion on native C++.

The Managed Heap

When you call new on a C# class, the class data is allocated sequentially in a continuous block in the managed heap. When the CLR calculates that you have no more references to an object, the object becomes a candidate for garbage collection.

Over time, multiple object allocations and orphaned objects can cause a single large continuous block of free memory to become fragmented into allocated memory and bubbles of unreferenced memory. A subsequent call to the allocation mechanism might not be able to find a sufficiently large contiguous block of memory to contain the new data, even though the total free memory in the system is larger than the desired amount. In this case, the CLR is able to collect the garbage and rearrange memory within the managed heap. In this process, similar to defragmenting a hard disk, in-use memory is moved to combine available memory bubbles to create larger contiguous blocks of memory. It is called *garbage collection*, because available bubbles of memory are not valid data, they are effectively "garbage," and combining available bubbles involves essentially collecting the garbage.

The Stack

In the CLI, the other primary memory buffer used for dynamic allocation of data is the program stack. A *stack* is a memory buffer that grows in a single direction and shrinks in the opposite direction. New allocations can only be made at the top of the stack, and only the memory at the top of the stack may be freed. Allocating memory on the stack is called *pushing*, and freeing memory off the stack is called *popping*. In computer science parlance, the stack is a First-In, Last-Out (FILO) buffer, which means that the first data you push onto the stack is the last data you pop off.

At first glance, using a stack seems to be overly restrictive and not as useful as you might like. In reality, a stack is particularly useful for making function calls and essential for recursive calls. All processors today are speed optimized for using stacks. In C# and C++, the program stack is where return addresses for procedure calls are stored as well as value types and the handles to the reference types. Because of the way that stacks are allocated and deallocated, stacks never become fragmented, and garbage collection is never necessary, thus the restrictive nature of a stack realizes performance benefits.

The Native Heap

Native C++ has a third and final memory area for dynamic allocation called the native heap. Allocations on the native heap are made using the new keyword. C++/CLI applications can use the native heap as well as the managed heap and stack for memory allocation. We will discuss this further in Chapter 18.

Garbage Collection

Recall that when the managed heap becomes fragmented, objects must be moved around to create larger contiguous blocks of memory in a process called *garbage collection*. As will be discussed further in Chapter 20, specific instances of reference types may be excluded from garbage collection by a process known as *pinning*, but this can have negative performance implications.

Let me reiterate: In-use memory is moved. This means that if you have a reference type in your program, it may be moved without your knowledge.

When you use a reference type, it is made of two parts: the data itself, which is allocated on the managed heap, and a handle to the data, which is allocated on the stack. We will revisit the stack in greater detail later in this chapter, but for now, suffice it to say that the stack does not move in the same way.

When garbage collection is performed, the data in the managed heap is moved to make larger contiguous blocks free for allocation, and at the same time, any handles that point to this data must continue to point to this data after garbage collection is complete. If you like, you can think of handles as pointers, and visualize the pointers to the instances of data on the managed heap being updated each time the data is moved. It does not matter how this is actually implemented within the CLR, but if the garbage collection mechanism is working correctly, your handles will continue to track your data after garbage collection is complete.

Initialization

As I mentioned previously, C# hides the implementation differences between reference types and value types. Consider the following C# example:

```
struct V
{
}
class R
{
    static public void Main()
    {
        V v = new V();
        R r = new R();
    }
}
```

In this example, we have a simple value type `V` and a reference type `R`. The procedure `Main()` is a public static function that allocates a `V` as well as an `R`. When you compile this example code and examine the resulting executable using `ildasm.exe`, you find the following CIL within the `Main()` method:

```
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       16 (0x10)
  .maxstack  1
  .locals init (valuetype V V_0, class R V_1)
  IL_0000:  nop
  IL_0001:  ldloca.s   V_0
  IL_0003:  initobj      V
  IL_0009:  newobj    instance void R::.ctor()
  IL_000e:  stloc.1
  IL_000f:  ret
} // end of method R::Main
```

As you can see from the CIL, the value type `V` is initialized with the `initobj` instruction, which initializes `V` on the stack. `initobj` is used to initialize a value type when there is no constructor. The reference type `R` is initialized by the `newobj` instruction, which calls `R`'s constructor, allocates `R`'s data on the managed heap, and returns a handle to this data. These are extremely different operations.

The C++/CLI Equivalent

Let's look at the equivalent code in C++/CLI:

```
value class V
{
};
ref class R
{
    static public void Main()
    {
        V v = V();
        R^ r = gcnew R();
    }
};
```

As you can see, `gcnew` is not used when `v` is allocated, which makes sense, because we don't want to allocate `v` on the managed heap. It is allocated on the stack, and the C++/CLI code reflects that. The CIL also reflects this, as it uses `initobj` rather than `newobj` to instantiate `v`. Granted, `gcnew` could have been used to allocate an instance of `V` on the managed heap. This operation is called *boxing*. We discuss boxing later in this chapter. For the sake of this example, we want to allocate it on the stack.

We can see from this simple example that where C# attempts to hide the implementation from the user to ease programming, C++/CLI remains loyal to the implementation and maps directly to the CIL.

Uninitialized Declarations

C# has declaration syntax for uninitialized value types but requires them to be initialized before they can be used. Consider the following C# code:

```
struct V
{
    public int i;
}
class R
{
    static public void Main()
    {
        V v;
        System.Console.WriteLine(v.i);
    }
}
```

If you attempt to compile this, you get the following error:

```
h.cs(10,34): error CS0170: Use of possibly unassigned field 'i'
```

C# blocks you from using uninitialized memory. Similar syntax in C++/CLI produces different results:

```
private value class V
{
public:
    int i;
};
private ref class R
{
public:
    static void Main()
    {
        V v;
        System::Console::WriteLine(v.i);
    }
};
```

This seemingly similar code compiles and runs without error and produces the following result:

```
0
```

Let's pass it through .NET Reflector to look at `Main()` and figure out what code the C++/CLI compiler generates. Figure 6-1 shows .NET Reflector's view of the code.

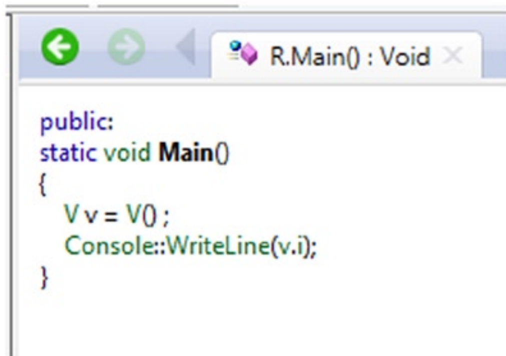

Figure 6-1. *.NET Reflector's view of the translation of an uninitialized declaration to C++/CLI*

As you can see from the figure, C++ does in fact initialize `v`, and running the program produces `0` as a result of the default initialized value of `int` being zero.

Initialization Variants

Let's summarize the preceding discussion by using .NET Reflector to analyze the following code:

```
value struct V
{
    V(int i)
    {
    }
};
ref struct R
{
    static public void Main()
    {
        V v1;
        V v2(1);
        V v3 = V(2);
    }
};
```

Figure 6-2 shows what the compilers generate. Note that the variables are renamed in the IL.

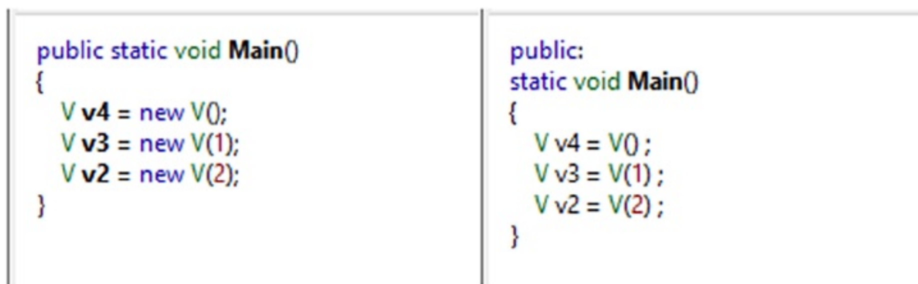

Figure 6-2. *Initialization of value types in C++/CLI*

As you can see from the figure, all variants of V are initialized in some manner by C++.

Fields

Consider the following C# code:

```
using System.Collections;
class R
{
    ArrayList a = new ArrayList();
    static void Main() {}
}
```

Although this code seems quite normal, it does not map directly to C++/CLI. This is because the variable a is a non-static field and needs to be initialized every time a class is instantiated. Therefore C# implicitly creates a constructor that undertakes the responsibility of initializing the instance fields. This does not map directly to C++/CLI, but it is easily emulated.

Let's employ .NET Reflector to see the constructor C# generated for this snippet; Figure 6-3 shows the constructor.

Figure 6-3. *The implicitly generated constructor*

As you can see, a constructor was generated that initializes the variable a. If we now switch to C++/CLI mode, we can view the constructor we need to write to convert this snippet, as shown in Figure 6-4.

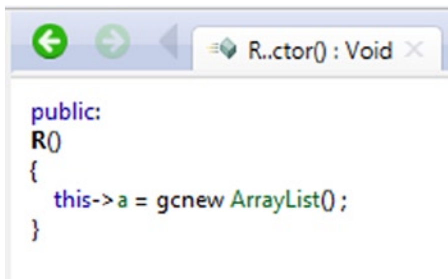

Figure 6-4. *The C++/CLI version of the constructor*

We can also use Instant C++ to aid translation. This is the output from the Snippet Converter:

```
//.h file code:

class R
{
private:
	ArrayList *a;
	static void Main();

private:
	void InitializeInstanceFields();
```

```
public:
        R()
        {
                InitializeInstanceFields();
        }
};

//.cpp file code:

void R::Main()
{
}

void R::InitializeInstanceFields()
{
        a = new ArrayList();
}
```

In this case, Instant C++ automatically creates an initialization function that is called by the class constructor.

Note that this version of the C++/CLI Reflector add-in does not qualify `ArrayList` with the namespace `System::Collections`. We don't need to use this if we have the `using` statement in the code, as follows:

```
using namespace System::Collections;
```

We now can use this accumulation of knowledge to craft the conversion of the C# code:

```
using namespace System::Collections;
ref class R
{
    ArrayList ^a;
    static void Main() {}
public:
    R()
    {
        this->a = gcnew ArrayList();
    }
}
```

Multiple Constructors

As you learned previously, we need to move object initialization code into the class's constructor. What happens if the class has more than one constructor? What does C# do?

Consider the following C# snippet:

```
class R
{
    class R1 {}
    R1 rA = new R1();
    R(int i) {}
    R() {}
    static void Main() {}
}
```

In this case, `rA` needs to be initialized once for each object, and two different constructors are available. See Figure 6-5 to look at these constructors using .NET Reflector.

C#

```
private R()
{
    this.rA = new R1();
}
```

C++/CLI

```
private:
R()
{
    this->rA = gcnew ::R1() ;
}
```

```
private R(int i)
{
    this.rA = new R1();
}
```

```
private:
R(int i)
{
    this->rA = gcnew ::R1() ;
}
```

Figure 6-5. *Class initialization with multiple constructors*

As you can see from the figure, C# copies the initialization code to both constructors independently.

Static Initialization

Let's now consider a broader example that builds on the previous examples by using both static and normal initializations. Consider the following C# code:

```
class R
{
    class R1
    {
    }
    struct V1
    {
    }
    V1 vA = new V1();
    R1 rA = new R1();
    V1 vB;
    R1 rB;
    static V1 vC = new V1();
    static R1 rC = new R1();
    R()
    {
        vB = new V1();
        rB = new R1();
    }
```

```
    static public void Main()
    {
        R r = new R();
    }
}
```

Even though this class already has a constructor, C# still moves initializations around for the CIL to operate correctly. We can employ .NET Reflector to discover which initializations were moved during compilation, which guides us on how to make an equivalent C++/CLI program. Figure 6-6 shows the members of class R as displayed in .NET Reflector.

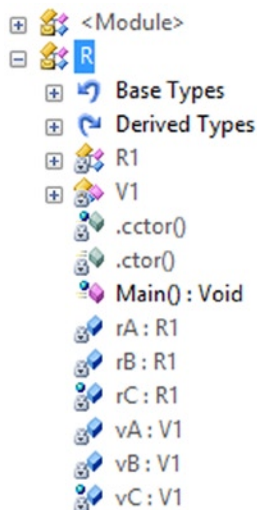

Figure 6-6. *Class R in .NET Reflector*

As you can see, not only does `R` have a constructor, which is represented as `.ctor`, it also has a static constructor, which is represented as `.cctor`.

Static Constructors

A *constructor*, more verbosely called an *instance constructor*, is called each time an instance of a class is created. A *static constructor*, also known as a *class constructor* or a *type initializer*, is called only once, before any instances of the class are created. It is used for one-time initialization of data common to all instances.

Now let's get back to examining the code and peek at the constructor and static constructor. Both of these constructors are relocation targets of the C# compiler. The instance constructor gets all instance initializations, and the static constructor gets all static initializations. First let's examine the constructor shown in Figure 6-7.

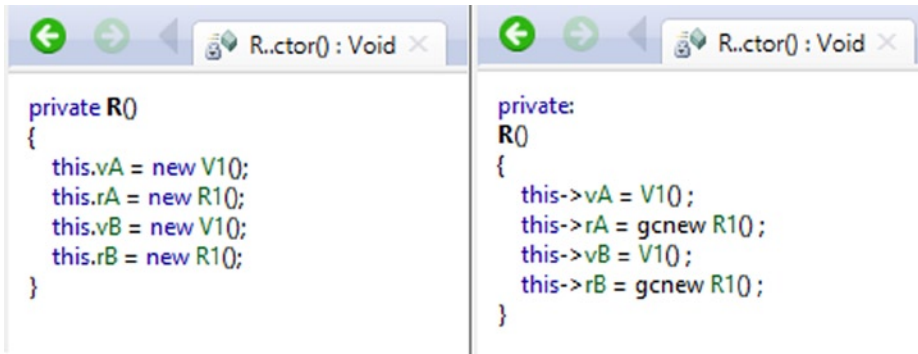

Figure 6-7. *Class R's constructor*

Similar to what you learned previously, the initializations for `vA` and `rA` are moved to the constructor. No surprises there. How about the static constructor shown in Figure 6-8?

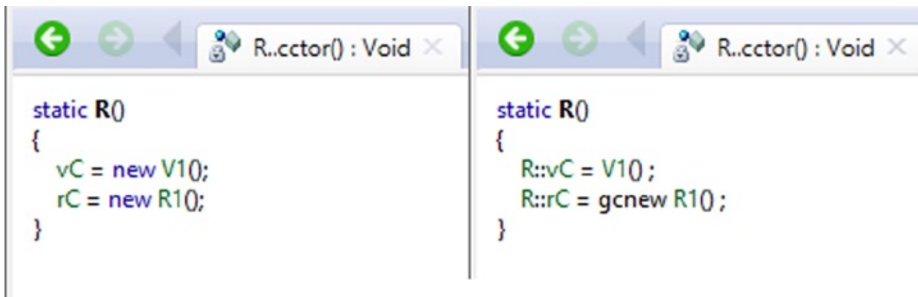

Figure 6-8. *Class R's static constructor*

Similar to the move of `vA` and `rA`, `vC` and `rC` are moved to the static constructor. This makes sense as well, because if regular initializations are moved to the constructor, then static initializations should be moved to the static constructor. C++/CLI moves the static initializations to the static constructor automatically, so we can let the compiler create it implicitly. C++/CLI can do this, because there is at most one static constructor in a class, though there may be multiple instance constructors in a class.

Now we can construct the final C++/CLI code and finish this aspect of the topic of initialization:

```
ref class R
{
    ref class R1
    {
    };
```

```
    value class V1
    {
    };
    V1 vA;
    R1 ^rA;
    V1 vB;
    R1 ^rB;
    static V1 vC = V1();
    static R1 ^rC = gcnew R1();
    R()
    {
        vA = V1();
        rA = gcnew R1();
        vB = V1();
        rB = gcnew R1();
    }
public:
    static void Main()
    {
        R ^r = gcnew R();
    }
};
```

Alternatively, in C++/CLI, vC and rC could be initialized using an explicit static constructor as follows:

```
private:
    static R()
    {
        vC = V1();
        rC = gcnew R1();
    }
```

The static constructor is executed before any instantiations of the class R are performed. Consider the following application:

```
using namespace System;
ref struct R
{
    static R()
    {
        Console::WriteLine('Static Constructor');
    }
    R()
    {
        Console::WriteLine('Constructor');
    }
};
```

```
int main()
{
    R ^r;
    Console::WriteLine('in main()');
    r = gcnew R();
}
```

This program has the following output:

```
Static Constructor
in main()
Constructor
```

This output shows that the static constructor for R was called before any R objects were instantiated.

Boxing

Because there are two major categories of types in .NET, value and reference, it is not surprising that we occasionally need to perform some type of conversion between them.

Oftentimes, we need to pass a value type to a method that requires a reference type. This task is potentially daunting, as value types are stored on the stack, whereas reference types are stored on the managed heap. Java has wrapper classes to solve this kind of problem; the CLR provides boxing.

The process of couching a value type as a reference type is called *boxing*. Boxing returns a `System::Object^` that references a copy of the value data and can be used to refer to the data while allocated on the managed heap. Boxing is generally automatic and implicit.

The inverse operation, retrieving the original value type from the boxed object is called *unboxing*. Unlike boxing, unboxing must be done explicitly. This is intuitive, because all value types become a single boxed object, so the CLR knows exactly what to do. On the other hand, given a boxed object, the CLR cannot determine which value type is contained therein without an explicit cast.

Method Boxing

Many CLR methods accept reference types as parameters. For example, `Console::WriteLine()` accepts either a built-in type or a reference type as parameter.

Example of Boxing and Unboxing

Consider the following example of boxing and unboxing. In this sample, we take a value type `V`, box it into an `Object`, and send it to `Console::WriteLine()` as an `Object`. Next, we unbox it explicitly to a `V` and send it to `Console::WriteLine()` again, which boxes it implicitly. Therefore, the following example contains both implicit and explicit boxing, as well as explicit unboxing:

```
using namespace System;
value struct V {};
ref struct R
```

```
{
    static void Main()
    {
        V v;
        Object ^o = v;
        Console::WriteLine(o);
        v = (V) o;
        Console::WriteLine(v);
    }
};

int main()
{
    R::Main();
}
```

The results of this program follow:

```
V
V
```

Delving into the IL, we can see the CIL boxing operations:

```
.method public hidebysig static void Main() cil managed
{
      // Code Size: 47 byte(s)
      .maxstack 1
      .locals (
            V v1,           //this is ldloc.0
            object obj1)    //this is ldloc.1
      L_0000: ldnull        // 0
      L_0001: stloc.1       // obj1 = 0
      L_0002: ldloca.s v1   //
      L_0004: initobj V     // v1 = V()
      L_000a: ldloc.0       // get v1
      L_000b: box V         // box it (explicit)
      L_0010: stloc.1       // obj1 = boxed(v1)
      L_0011: ldloc.1       // get obj1
      L_0012: call void [mscorlib]System.Console::WriteLine(object)
      L_0017: ldloc.1       // get obj1
      L_0018: unbox V       // unbox obj1 of type V
      L_001d: ldobj V       // get V
      L_0022: stloc.0       // v1 = unboxed
      L_0023: ldloc.0       // get v1
      L_0024: box V         // box it (implicit)
      L_0029: call void [mscorlib]System.Console::WriteLine(object)
      L_002e: ret
}
```

You don’t need to be an expert in CIL to see what is going on here, especially since I have annotated the individual instructions.

Unboxing Dangers

Because unboxing is explicit, there is the danger of a programmer unboxing an object to the wrong type, which generally causes the CLR to throw an exception. Consider the following example:

```
using namespace System;
using namespace System::Collections;
ref struct R
{
    static void Main()
    {
        ArrayList^ a = gcnew ArrayList();
        int i=3;
        double d=4.0;
        a->Add(i);
        a->Add(d);
        for each(int j in a)
        {
            Console::WriteLine(j);
        }
    }
};
void main() { R::Main();}
```

In this example, we implicitly box an `int` and a `double` by adding them to an `ArrayList()`. The `for each` loop unboxes these values into an `int`, causing an exception when the `double` is unboxed. The results to the screen follow:

```
Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
   at R.Main()
   at mainCRTStartup(String[] arguments)
```

Safe Unboxing

In C#, you can fix the preceding code by using the keyword `is` as follows:

```
using System;
using System.Collections;
class R
{
    public static void Main()
    {
        ArrayList a = new ArrayList();
        int i = 3;
        double d = 4.0;
        a.Add(i);
        a.Add(d);
        foreach(Object o in a)
```

```
        {
            if(o is int)
            {
                int i1 = (int)o;
                Console.WriteLine(i1);
            }
            else
            {
                double d1 = (double)o;
                Console.WriteLine(d1);
            }
        }
    }
}
```

In this code, you can see the check to see whether the object is a boxed `int` (`o is int`). To perform the same trick in C++/CLI, you use `dynamic_cast<>()` as follows:

```
using namespace System;
using namespace System::Collections;
ref struct R
{
    static void Main()
    {
        ArrayList^ a = gcnew ArrayList();
        int i=3;
        double d=4.0;
        a->Add(i);
        a->Add(d);
        for each(Object ^o in a)
        {
            if(dynamic_cast<int^>(o) != nullptr)
            {
                int i1=(int)o;
                Console::WriteLine(i1);
            }
            else
            {
                double d1=(double)o;
                Console::WriteLine(d1);
            }
        }
    }
};
void main() { R::Main();}
```

The casting operators are revisited in more detail in Chapter 16.

Constructor Forwarding

C# has a special syntax that allows you to defer initialization of objects between constructors. This is called *constructor forwarding*, or delegating constructors, and is now supported in the current version of C++/CLI. Here is a C# example of constructor forwarding:

```
class R
{
    R(int i) {}
    R() : this(5) {}
    public static void Main() {}
}
```

Here is an example in C++/CLI, using slightly different syntax:

```
ref struct R
{
    int value;
    R(int i)
    {
        value = i;
    }
    R() : R(5)
    {
    }
};
void main()
{
    R^ r = gcnew R();
    System::Console::WriteLine(r->value);
}
```

Running this example displays the number 5.

In this example, the `R()` constructor forwards construction to the `R(int)` constructor and continues construction in the `R()` method.

Now beware, the following attempt is misguided:

```
ref struct R
{
    R(int i)
    {
    }
    R()
    {
        R(5);
    }
};
void main()
{
    R^ r = gcnew R();
}
```

This code does not work for the following reason: when the `R()` constructor calls `R(5)`, it does not instruct the compiler to forward construction to the `R(int)` constructor. Rather, it creates a temporary copy of `R`, initialized with `0`, which is discarded upon exit from the method. Figure 6-9 shows the `R()` constructor using .NET Reflector.

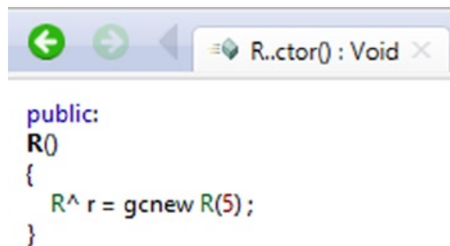

Figure 6-9. *A failed attempt at constructor forwarding in C++/CLI*

As you can see, the compiler creates a temporary of type `R` with a `5` as an initialization parameter, which does not accomplish the same thing as C# constructor forwarding.

C# Partial Classes

C# allows the definition of a single class to span multiple files using the `partial` keyword. C++/CLI has no direct analogue, as it supports a different paradigm. As alluded to previously, C++ allows you to place a class declaration in a single header file and implement the class members in multiple C++ files, but this has nothing to do with the concept of partial classes.

Reference Types on the Stack

A less-often-used feature of C++/CLI is the ability to declare and use a reference type as if it were a stack variable like a value class. This is strictly syntactic sugar and is not needed for standard C++/CLI programming. Even though the memory is still physically allocated on the managed heap, the compiler makes the object observe the semantics of an object allocated on the stack.

Basic Types

Let's look at Table 6-2, which contains the built-in C# types, to see how they map to C++/CLI.

Table 6-2. *Mapping Basic C# Types to C++/CLI*

C#	C++/CLI	Bytes	.NET Type	Signed	Marshal	Example
sbyte	char	1	SByte	Yes	No	-1, 'A'
byte	unsigned char	1	Byte	No	No	3u, 0xff
short	short	2	Int16	Yes	No	-1
ushort	unsigned short	2	UInt16	No	No	3u
int	int or long	4	Int32	Yes	No	-1l
uint	unsigned int or unsigned long	4	UInt32	No	No	3u, 3ul
long	long long	8	Int64	Yes	No	-1ll
ulong	unsigned long long	8	UInt64	No	No	3ull
single	float	4	Single	Yes	No	4.0f
double	double	8	Double	Yes	No	3.0
string	System::String^	N/A	String	N/A	No	"A"
object	System::Object^	N/A	Object	N/A	No	N/A
decimal	System:Decimal	16	Decimal	Yes	No	N/A
char	wchar_t	2	Char	No	Yes	L'A'
bool	bool	1	Boolean	N/A	Yes	true

Basic Type Differences

One important thing to note is that some of the common C# types have no analogue in C++/CLI or are slightly different in C++/CLI and require some level of marshaling for conversion.

Missing Keywords

The `string`, `object`, and `decimal` keywords in C# have no corresponding built-in type in C++/CLI. Does this mean we can't use these types in C++? Not at all. In fact, because both C# and C++/CLI target the CLI, we can always resort to specifying the CLI target type by name and using that instead.

Marshaling Required

What is marshaling? In general, a *marshaler* is a program that translates or packages data between two programs. A marshaler may be required in a number of circumstances where two programs cannot operate seamlessly on a single instance of data. The C++/CLI types `wchar_t` and `bool` have marshaling attributes attached to them in the metadata for proper consumption from C# and other .NET languages. Attributes will be discussed in greater detail in Chapter 20. Until then, let's look at some marshaling attributes in the metadata.

Consider the CIL for the following simple C++/CLI method. Note that Hello() takes a wchar_t as a parameter and returns a bool; both of these types are marshaled:

```
ref class R
{
    bool Hello(wchar_t ch)
    {
        return true;
    }
};
```

Figure 6-10 shows the C# view of the Hello() method using .NET Reflector.

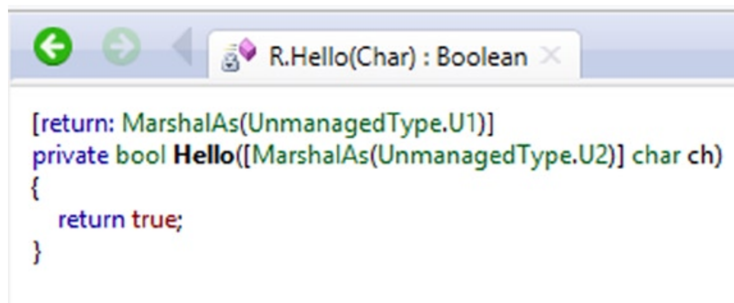

Figure 6-10. *C# view of Hello()*

As you can see, the MarshalAs(UnmanagedType.U1) return attribute is added to the bool return value, and MarshalAs(UnmanagedType.U2) is added to the char value (which corresponds to the wchar_t).

The UnmanagedType enum is a member of System::Runtime::InteropServices and indicates the kind of data being marshaled. If you want to see how the type is defined, you can find the definition in mscorlib.dll using .NET Reflector. You will discover that U1 is 4 and U2 is 6 not that it really matters! Now let's look at the C++/CLI version of Hello() using .NET Reflector shown in Figure 6-11.

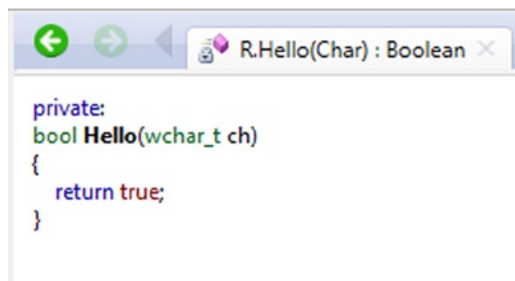

Figure 6-11. *C++/CLI view of Hello()*

Wait—where has the marshaling gone? Well, when we wrote the C++/CLI add-in, we realized that use of a wchar_t or a bool always emits a MarshalAs attribute, so these attributes are suppressed from the output. On the other hand, if you wanted to communicate something nonstandard, like marshaling a short as an unmanaged bool, as in the following:

```
using namespace System::Runtime::InteropServices;
ref class R
{
    [returnvalue: MarshalAs(UnmanagedType::Bool)]short Hello(wchar_t ch)
    {
        return (short)true;
    }
};
```

then the C++/CLI .NET Reflector add-in would not suppress the marshal attributes, because they differ from the default. We can see this in Figure 6-12.

Figure 6-12. *C++/CLI Hello() with special marshaling*

As you can see from the screenshot, the special marshaling we have added for our short conversion to unmanaged bool displays clearly in .NET Reflector.

Table 6-3 lists the various strengths and weaknesses of the C++ class types.

Table 6-3. *Feature Limitations by Class Type*

Feature	Native Class	Ref Class	Value Class	Interface
Assignment operator	X	X		
Class modifier	X	X	X	
Copy constructor	X	X		
Delegate definitions	X	X	X	X
Default constructor	X	X		
Destructor	X	X		
Events		X	X	X
Finalizer		X		

(*continued*)

Table 6-3. (*continued*)

Feature	Native Class	Ref Class	Value Class	Interface
Function modifiers	X	X	X	X
`initonly field`		X	X	X
Literal field		X	X	X
Member of delegate type		X	X	
Override specifier	X	X	X	
Parameter arrays	X	X	X	X
Properties		X	X	X
Reserved member names	X	X	X	X
Static constructor		X	X	X
Static operators	X	X	X	X

Summary

That concludes our introduction to classes and class types.

If you didn't understand everything in this chapter, that's okay—go on to the next one. You see, the goal of this chapter is not to drill in the details about the type system but to expose you to a little bit more of what is going on behind the scenes. We will continue this strategy as the book unfolds and come back to the important concepts again and again.

Next let's look at that all-important basic data structure, the array.