CHAPTER 3

Syntax

The pure and simple truth is rarely pure and never simple.

—Oscar Wilde

The previous chapters emphasized the similarities between C# and C++/CLI. Now we touch on the main areas where they differ and begin to understand why. These include additional or different keywords, separators, and operators.

Keywords and Separators

In C++, the additional keyword `namespace` is required when `using` a namespace (see Table 3-1).

Table 3-1. *Namespaces in C# and C++/CLI*

C#	C++/CLI
`using System.Threading;`	`using namespace System::Threading;`
`System.Console.WriteLine("H");`	`System::Console::WriteLine("H");`

Moreover, where C# uses dot as a general separator, C++ employs several different separators depending on the context as well as the meanings of the items being separated. The most common of these separators are colon-colon (`::`) and dot (`.`). The colon-colon separator, or scope resolution operator, is used to qualify identifiers with namespaces, classes, properties, and events and to access static fields and methods. In both languages, the dot separator, or member access operator, is used to access members of instances of classes.

The paradigms of C++, different separators in different contexts, and of C#, a single separator for all contexts, are consistent with the overall design philosophy of each of the languages. C# favors simplicity, whereas C++ demands a deeper level of specificity in exchange for greater flexibility.

Table 3-2 shows separator differences between C# and C++. I cover all these separators in detail as the book progresses.

Table 3-2. *Separators in C++*

Separator	Name	Meaning
`::`	colon-colon	Scope resolution operator, used when the expression to the left of the `::` is a namespace, class, property, or event name and the expression to the right of the `::` is a namespace, class name, or static member of a class. With no left-expression, the expression on the right is a global variable.
`.`	dot	Class member access operator, used when the expression to the left of the arrow is a class object
`->`	arrow	Class member access operator, used when the expression to the left of the arrow is a pointer or handle to a class object
`.*`	dot star	Pointer to a member operator, used when the expression to the left of the arrow is a class object and the expression to the right of the arrow is a pointer to a member of the same class
`->*`	arrow star	Pointer to a member operator, used when the expression to the left of the arrow is a pointer to a class object and the expression to the right of the arrow is a pointer to a member of the same class

C# and C++ define classes and structures differently. In addition to one obvious syntactic difference—C++ requires a trailing semicolon after a type definition—significant semantic differences exist. See Table 3-3 for an example comparing classes and structures in C# and C++.

Table 3-3. *Classes and Structures in C# and C++/CLI*

C#	C++/CLI
`class R {}`	`ref class R {};`
N/A	`ref struct R {};`
`struct V {}`	`value class V {};`
N/A	`value struct V {};`
`enum E {}`	`enum class E {};`
N/A	`enum struct E {};`
N/A	`class C {};`
N/A	`struct C{};`

In C#, classes and structures are vehicles for implementing *reference types* and *value types* as defined by the CLI. In C++, classes and structures define a *type*—in general, a related collection of fields and methods and subtypes.

C++/CLI introduces two class modifiers, `ref` and `value`, which provide a way to represent the CLI class types in C++. Together with the `class` or `struct` keyword and separated by whitespace, as in `ref class`, they form a single new keyword, appropriately called a *whitespace keyword*.

Reference types and value types are very important in .NET programming, and it's a good idea to review these types a bit before we continue. There are many practical differences between reference types and value types, but the main differences relate to how they are allocated. A reference type is allocated in two parts. A reference type's data is allocated on the managed heap, while a separate handle to this data is allocated on the stack. A value type is allocated automatically on the stack.

A C# `class` is a reference type; so is a C# `string`. A C# `struct` and most C# built-in types, including `int` and `char`, are value types. Value types contained in reference types, either explicitly or implicitly via boxing, become elements of the reference type and are allocated on the managed heap.

C# class (Reference Type)

Suppose you have a C# `class` named `Hello`. Allocate an instance using

```
Hello h = new Hello();
```

From the syntax, it appears that you have created a single unified entity of type `Hello`. Behind the scenes there is much more going on, as data was allocated on the stack as well as the managed heap. An instance of the `Hello` object was allocated on the managed heap, and a handle to this instance was stored on the stack in the variable `h`.

C# struct (Value Type)

If `Hello` is defined as a C# `struct`, then a completely different operation occurs. The entire instance of `Hello` is allocated on the stack, and `h` represents the instance of this object.

Caveat

The fact that reference types are divided between the stack and heap generates some interesting and somewhat unintuitive results when you're assigning values to reference types. When you assign one value type to another, you copy the data associated with one instance of the type to another instance. When you assign one reference type to another, you overwrite the handle to one instance with the handle of another instance. The instances themselves remain unchanged.

Consider the following code in C#:

```
class Hello
{
    int i;
    Hello(int number)
    {
        i=number;
    }
    static void Main()
    {
        Hello h = new Hello(1);
        Hello j = new Hello(2);
        j = h;
        System.Console.WriteLine(j.i);
        h.i = 3;
        System.Console.WriteLine(j.i);
    }
}
```

After compiling and running this code, we get

```
C:\>csc /nologo test.cs
C:\>test
1
3
```

In this program, we allocate two objects of type `Hello` on the managed heap. The handles to these classes, `h` and `j`, are allocated on the stack. We overwrite the handle in `j` with the handle in `h` and orphan `Hello(2)`. `Hello(2)` becomes available for reclamation by the garbage collector. Both `h` and `j` now reference the `Hello(1)` object, and there is no difference between accessing the member field `i` using `h` or using `j`.

In other words, since `Hello` is a reference type, `h` and `j` are handles that point to data on the managed heap. When the assignment `j=h` occurs, `h` and `j` both refer to the same data. Assigning `3` to `h.i` also affects `j.i`, and displaying `j.i` results in the number `3`.

Contrast

On the other hand, if `Hello` were a value type, you would see a different result. Change the declaration of `Hello` from `class` to `struct`:

```
struct Hello
{ /**/ }
```

After compiling and executing the program, we see

```
C:\>csc /nologo test.cs
C:\>test
1
1
```

The results are different this time, because our objects are all allocated on the stack and are overwriting one another.

Lack of Locality

A local inspection of the method `Main()` is insufficient to determine the results of the program. You cannot determine what result the `WriteLine` will generate by just looking at the surrounding code. C# requires you to refer to the definition of `Hello` and discover whether `Hello` is a `class` or a `struct`.

This lack of locality is dangerous and goes against the C++/CLI design philosophy. In C++/CLI, the distinction between reference types and value types is much more explicit. The programmer specifies more precisely what he or she wants to do, which avoids confusion and ultimately makes the code more maintainable. The cost is that the syntax is slightly more difficult.

The C++ Approach

In C++/CLI, handles are typically flagged using the handle punctuator `^`. It is also called a *tracking handle*, because it points to an object that may be moved around during garbage collection.

Translating the previous code to C++/CLI, we achieve the following:

```
private ref class Hello
{
private:
    int i;
    Hello(int number)
    {
        i=number;
    }
public:
    static void Main()
    {
        Hello ^h = gcnew Hello(1);
        Hello ^j = gcnew Hello(2);
        j = h;
        System::Console::WriteLine(j->i);
        h->i = 3;
        System::Console::WriteLine(j->i);
    }
};
void main()
{
    Hello::Main();
}
```

After compiling and executing, we get

```
C:\>cl /nologo /clr:pure test.cpp
C:\>test
1
3
```

There are a few obvious syntactic differences from the C# version. However, I'd like to start off by pointing out a semantic difference. In C++/CLI, changing `Hello` from a reference type to a value type, by changing the whitespace keyword `ref class` to `value class`, does not produce different results on compilation and execution.

Changing the type from a reference type to a value type affected where the type was allocated, but it did not change the fact that in the previous code snippet we are treating the data as referenced data. If `Hello` morphs into a value type, then the compiler generates different IL, so that `h` and `j` remain handles to the data on the managed heap, and the result is consistent. Behind the scenes, the value types are boxed—we'll revisit that later in Chapter 6.

Types of Member Access Operators

The other important difference between the C++ snippet and the C# snippet is that C++ handles use a different class member access operator. The syntax is similar to that of pointers in C++, as handles may be considered a special kind of pointer. If you are working with a handle or pointer to an object, you use the arrow member access operator (`->`) to access the object's members. If you are working with an instance of the object itself, you use the dot member access operator (`.`). Although it may seem more complicated to have two different types of member access operators, one benefit is that code like our previous example always does what you expect it to, because you are forced to be mindful of what you're doing as you write—and that's a good thing.

Keyword Differences

In this section, we go over the keyword differences between C# and C++. Most of these differences are because of the evolution of the C++ language and the compatibility and disambiguation restrictions for adding to the C++ grammar.

Let's begin with the `foreach` keyword, shown in Table 3-4.

Table 3-4. *foreach in C# and for each in C++/CLI*

C#	C++/CLI
`foreach`	`for each`

In C++/CLI, the keyword `for each` has a space, and the usage differs slightly from `foreach` in C#. The converted code appears in Table 3-5.

Table 3-5. *Examples of foreach in C# and for each in C++/CLI*

C#

```
using System;
using System.Collections;
class R
{

    static void Main()
    {
        ArrayList list = new ArrayList(0);
        list.Add("hello");
        list.Add("world");
        foreach (Object o  in list)
        {
            Console.WriteLine(o);
        }
    }
}
```

C++/CLI

```
using namespace System;
using namespace System::Collections;
ref class R
{
public:
    static void Main()
    {
        ArrayList ^list = gcnew ArrayList(0);
        list->Add("hello");
        list->Add("world");
        for each (Object ^o  in list)
        {
            Console::WriteLine(o);
        }
    }
};
void main()
{
    R::Main();
}
```

Review

Let's review what you've seen so far. Differences between C# and C++/CLI include the following:

- The additional keyword `namespace` is used.
- Namespaces are separated by colon-colon (`::`) instead of a dot (`.`).
- `ref class` is used instead of `class`.
- The punctuator `^` is used to declare handles.
- An arrow (`->`) is used as a handle member access operator, not a dot (`.`).
- `for each` contains a space.
- The class definition ends with a semicolon (`;`).
- C++/CLI begins programs with a global function named `main()`.

Now let's continue on; you can see that C++/CLI uses the keyword `nullptr` instead of `null` in Table 3-6.

Table 3-6. *null and nullptr*

C#	C++/CLI
`null`	`nullptr`

These keywords are used as shown in Table 3-7.

Table 3-7. *Usage of null and nullptr*

C#	C++/CLI

`class R`
`{`
`  static void Main()`
`  {`
`  R r = null;`
`  }`
`}`
	`ref class R`
	`{`
	`  static void Main()`
	`  {`
	`  R ^r = nullptr;`
	`  }`
	`};`

There are significant differences between `switch` and `goto` in C# and C++, as introduced in Table 3-8.

Table 3-8. *switch, case, and goto in C# and C++*

C#	C++
Does not allow case statements to fall through	Allows case statements to fall through
`goto` *case_statement*	N/A
`goto` *label*	`goto` *label*
`switch(string s)`	N/A

In C#, if a `break` or `goto` is missing from a nonempty case statement, the compiler issues an error. In C++, execution is said to *fall through* from a case to the case below it and continue with the next case.

Both languages support a `goto` keyword to a user-defined label. C# allows an explicit `goto` to a case statement. There is no C++ equivalent, and the reason is largely historical. In C, a `switch/case/break` construct was not so much a formal fork as a macro replacement for `goto`. The cases are not distinct blocks, but rather labels that act as switch targets. C switches were modeled after assembly language jump tables. C++ retains its heritage. C# attempts to employ a more formal abstraction, where the cases are truly distinct and disconnected entities, so C# naturally does not support fall through. Both abstractions have their respective advantages and disadvantages.

The C# construct `switch(string)` is not supported in C++. In C++, you must expand your `switch` statement using `if` and `else`. See Table 3-9 for example uses of `switch` in `goto` and fall through cases in C# and C++.

Table 3-9. *Usage of switch in C# and C++*

C#	C++

`// switch on a System.String and goto case`
`string s="1";`
`switch(s)`
`{`
`  case "1":`
`  goto case "2";`
`  case "2":`
`  break;`
`}`
`// fall through case not available`
	`// equivalent to switch on a System::String`
	`System::String ^s="1";`
	`if(s=="1")`
	`{`
	`}`
	`else if(s=="2")`
	`{`
	`}`
	
	`// fall through case`
	`int i,j=0;`
	`switch(i)`
	`{`
	`  case 1:`
	`  j++;`
	`// no break, so case 1 falls into case 2`
	`  case 2:`
	`  break;`
	`}`

Arrays and Functions

Managed arrays are declared differently in C++/CLI (see Table 3-10).

Table 3-10. *Managed Arrays in C# and C++/CLI*

C#	C++/CLI
`reftype []`	`array<reftype^>^`
`valuetype []`	`array<valuetype>^`
`class R`	`ref class R {};`
`{`	
`  static void Main()`	`void main()`
`  {`	`{`
`  R[] n = new R[5];`	`  array<R^> ^n = gcnew array<R^>(5);`
`  int[] m = {1, 2, 3, 4};`	`  array<int> ^m = {1, 2, 3, 4};`
`  m[3]=0;`	`  m[3]=0;`
`  }`	`}`
`}`	

Although they both are implemented using `System::Array`, C++/CLI uses a pseudo-template syntax for their declaration. Managed arrays will be explained in further detail in Chapter 7. Pseudo-template syntax is consistent with the way extensions have been added to the C++ language in the past, such as for the cast operators (see Chapter 16).

In both C# and C++, you can attach modifiers to function arguments. C# and C++/CLI pass parameter arrays, reference parameters, and out parameters differently, as shown in Table 3-11.

Table 3-11. *Function Argument Modifiers*

C#	C++/CLI
`params T[]`	`... array<T> ^`
`ref`	`%`
`out`	`[System::Runtime::InteropServices::Out] %`

We will revisit these later.

The Conversion Operators

The operations performed by the C# operators `is` and `as` may be performed by the C++ pseudo-template casting operators `static_cast<>()` and `dynamic_cast<>()` (see Table 3-12).

Table 3-12. *C# and C++/CLI Conversion Operators*

C#	C++/CLI
`as`	`dynamic_cast<>()`
`as`	`static_cast<>()`
`is`	`(dynamic_cast<>()!=nullptr)`

Conversion operators will be explained in further detail in Chapter 16.

Memory Allocation

The `new` operator indicates allocation on the native heap in C++. The `gcnew` operator was added in C++/CLI to indicate allocation on the managed heap. C# also uses the `new` operator to allocate value types on the stack. In C++, this is unnecessary, as the C++ syntax for allocating instances of user-defined value types is identical to the syntax for built-in types such as `int`. See Table 3-13 for a list of keywords used in allocation on the managed heap.

Table 3-13. *Allocation on the Managed Heap in C# and C++/CLI*

C#	C++/CLI
`new` (reference types)	`gcnew`
`new` (value types)	No operator is necessary

A short example of memory allocation on both the native and managed heaps in C++/CLI follows:

```
value struct V {}; //value type
ref struct R {};   //reference type
struct N {};       //native type
void main()
{
    N n;
    N *pN = new N();
    R ^r = gcnew R();
    V v;
}
```

Memory allocation will be discussed in further detail in Chapter 6.

Accessibility and Visibility

The accessibility and visibility keywords are similar, but the syntax is different. The keyword differences are listed in Table 3-14, and the syntactic differences will be explained in detail in Chapter 8.

Table 3-14. *Basic Protection Mechanisms*

Type Attributes	C#	C++/CLI
Public	`public`	`public:`
NotPublic	`private`	`private:`
Assembly	`internal`	`internal:`
Family	`protected`	`protected:`
FamilyOrAssembly	`internal protected`	`protected public:`
FamilyAndAssembly	N/A	`protected private:`

Properties, Events, and Delegates

In Chapter 10, we will discuss properties, events, and delegates, but see Table 3-15 for an introduction.

Table 3-15. *Simple Example of a Property in C# and C++/CLI*

C#

```
class R
{

    private int V;

    public int Value
    {
        get
        {
            return V;
        }
        set
        {
            V = value;
        }
    }
}
```

C++/CLI

```
ref class R
{
private:
    int V;
public:
    property int Value
    {
        int get()
        {
            return V;
        }
        void set(int newV)
        {
            V = newV;
        }
    }
};
```

Generics

In Chapters 14 through 16, you will learn about generics and templates, but see Table 3-16 for an introduction.

Table 3-16. *Simple Example of a Generic in C# and C++/CLI*

C#	C++/CLI
`public class R<T>`	`generic <typename T> public ref class R`
`{`	`{`
	`private:`
`    private T m_data;`	`    T m_data;`
	`public:`
`    public R(T data)`	`    R(T data)`
`    {`	`    {`
`        m_data = data;`	`    m_data = data;`
`        System.Console.WriteLine(m_data);`	`    System::Console::WriteLine(m_data);`
`    }`	`    }`
`}`	`};`
`public class R1`	
`{`	
`    static void Main()`	`int main()`
`    {`	`{`
`        R<int> r = new R<int>(3);`	`    R<int> ^r = gcnew R<int>(3);`
`    }`	`}`
`}`	

Built-in Types

C# and C++/CLI map to the CLI types with different keywords, and the C++/CLI mappings are consistent with native C++ to the extent possible. See Table 3-17 for an introduction before we go into greater detail in Chapter 6.

Table 3-17. *Built-in Types*

C#	C++/CLI
byte	char
sbyte	signed char
short	short
ushort	unsigned short
int	int, long
uint	unsigned int, unsigned long
long	long long
ulong	unsigned long long
single	float
double	double
string	System::String^
object	System::Object^
decimal	System:Decimal
char	wchar_t
bool	bool

Summary

Although the sheer volume of differences between C# and C++ may seem daunting at first, after a while a pattern emerges. Each language is intelligently designed and internally consistent, and C++ syntax will become intuitive quite soon. In the next chapter, we will apply what we're learning by converting a C# program to C++/CLI line by line.