Programming in Visual C++ .NET™

Rex Jaeschke

Programming in Visual C++ .NET

Edition: 2.0

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Most of the information in this book was first published as a series called "C++/CLI by Example", in the *C/C++ Users Journal*, in 2004 and 2005.

Preface

Welcome to the world of C++/CLI,[1] an important new dialect of Standard C++. In this book, we'll see how C++ has been extended to allow it to exploit the CLI[2] platform.

The intended audience of this book is experienced C++ programmers who are faced with "getting up to speed" with C++/CLI, or who simply want to understand where Visual C++ is headed. It is not intended directly for those making the transition from Microsoft's earlier "Managed Extensions to C++" effort.[3]

Reader Assumptions

This is *not* a first course in C++, and I assume that you know how to use Visual C++.

To fully understand and exploit the material, you should be conversant with the following C++ topics:

- Header usage
- All the built-in types
- Basic I/O
- Enumerated types
- Data and function pointers
- Class design and implementation, including inheritance

Limitations

This book covers the .NET-related extensions in Visual C++. It also introduces the .NET class library. However, a very small percentage of that library's facilities are mentioned or covered in any detail. The .NET library contains so many functions that whole books have been written about that subject alone.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming language training for some 14 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and focused. Many books use GUI and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

---

[1] CLI stands for "Common Language Infrastructure", the subset of .NET that was standardized by Ecma Technical Committee TC39/TG3, and adopted by ISO/IEC.

[2] .NET is the name of a Microsoft product that is a superset of the CLI standard. Another implementation of the CLI is Mono, from Novell/Ximian, which runs on Windows and Linux. See http://www.mono-project.com/about/index.html.

[3] For help with that topic, look for Stan Lippman's comprehensive articles at http://msdn.microsoft.com/visualc/.

Different styles work for different teachers and different students. I do not suggest that my approach is better than any other is; I simply know that my approach works well, and has formed the basis of my successful seminar business.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory, within which each program has its own subdirectory.

Each chapter contains exercises. For each exercise having a concrete solution, that solution is provided electronically in a directory tree named Labs, where each chapter has its own subdirectory, within which each program has its own subdirectory.[1] Exercises having no general solution require experimentation or research in an implementation's documentation.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

What You'll Need

Apart from a good dose of enthusiasm and time, you'll need a compiler,[2] so you can try things yourself. (You should also get a copy of the C++/CLI standard.[3])

Assemblies and Metadata

The traditional C++ compilation model involves compiling each source file separately to object form, then linking all object files together—along with library functions—to make an executable. The CLI model is quite different; it involves the creation and use of assemblies.

Simply stated, an *assembly* is the output from a single compilation, regardless of how many input source files are involved. If that output has an entry point (a `main` function, for example), it is an .exe file; if it does not, it's a .dll file. Any compilation that refers to something from outside the assembly being created, must access that dependent assembly. There is no header-like mechanism to promise what will ultimately be available at link-time. Such external information must be accessible during compilation by having the compiler "look inside" dependent assemblies.

An assembly contains *metadata*, which describes the types and functions contained therein; it also contains instructions in the Common Intermediate Language[4] (CIL), which Microsoft calls MSIL. These instructions can then be executed by the platform-independent Virtual Execution System (VES).

---

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.
[2] A free copy of Microsoft's Visual C++ Express Edition can be downloaded from http://www.microsoft.com/visualc.
[3] A free copy of this can be downloaded from http://www.ecma-international.org/publications/standards/Ecma-372.htm
[4] CIL and VES are part of the CLI standard, ECMA-335, which can be downloaded from http://www.ecma-international.org/publications/index.html.

The Status of C++/CLI

The first implementation of C++/CLI was the Beta 2 release of Microsoft's Visual Studio .NET, in May 2005. That implementation was based on the draft C++/CLI standard produced by Ecma[1] Task Group TC39/TG5. (Task Group TG3 is responsible for the CLI standard.) TG5 started work on this standard in December 2003, and completed its work in September 2005. Its work was adopted by Ecma as a standard in December 2005.

This author serves as project editor of TG5 (and TG3).

Rex Jaeschke, September 2009

[1] Ecma is an international standards organization (http://www.ecma-international.org).

1. Getting Started

In this chapter, we'll define a relatively simple CLI type and use it from an application. We'll also see how to build the two corresponding project types in Visual C++.

1.1 Our First CLI Type

Let's look at the source code of a class that models a two-dimensional point (see directory gs01\Point).

```
/*1*/
using namespace System;
/*2*/
public ref class Point
{
        int x;
        int y;

public:
// define read-write instance properties X and Y

/*3a*/  property int X
        {
/*3b*/          int get() { return x; }
/*3c*/          void set(int val) { x = val; }
        }

/*4a*/  property int Y
        {
/*4b*/          int get() { return y; }
/*4c*/          void set(int val) { y = val; }
        }

// define instance constructors

/*5a*/  Point()
        {
/*5b*/          X = 0;
/*5c*/          Y = 0;
        }
```

```
/*6a*/  Point(int xor, int yor)
        {
/*6b*/          X = xor;
/*6c*/          Y = yor;
        }

// define instance functions

/*7a*/  void Move(int xor, int yor)
        {
/*7b*/          X = xor;
/*7c*/          Y = yor;
        }

/*8a*/  virtual bool Equals(Object^ obj) override
        {
/*8b*/          if (obj == nullptr)
                {
                        return false;
                }

/*8c*/          if (this == obj)     // are we testing against ourselves?
                {
                        return true;
                }

/*8d*/          if (GetType() == obj->GetType())
                {
/*8e*/                  Point^ p = static_cast<Point^>(obj);
/*8f*/                  return (X == p->X) && (Y == p->Y);
                }
                return false;
        }

/*9*/   virtual int GetHashCode() override
        {
                return X ^ (Y << 1);
        }

/*10a*/ virtual String^ ToString() override
        {
/*10b*/          return String::Concat("(", X, ",", Y, ")");
        }
};
```

Various source lines (or blocks of lines) are labeled with comments of the form `/*1*/`, `/*2*/`, `/*3a*/`, and so on. These shall be referred to as *cases*, as in case 1, case 2, case 3a, and so on.

1.1.1 Namespaces

All CLI standard library types reside in namespace `System` or in namespaces nested inside that one. Examples are the types `System::Object` and `System::String`, and the namespaces `System::IO`, `System::Text`, and `System::Runtime::CompilerOptions`. Case 1 avoids the need for using namespace qualification. For example, in case 10a, we can write `String` instead of its fully qualified name, `System::String`.

1.1.2 Defining a Ref Class

In case 2, we define a ref class called `Point`. A *ref class* is a CLI reference type. When taken together, `ref` and `class`, with intervening white space, make up a new keyword.[1]

The `public` prefix indicates that this type is visible outside its parent assembly. There are two kinds of *visibility*, public and private. By default, types have private visibility. Only types can have visibility; as such, non-member functions, global variables, and file-scope typedefs cannot be made visible outside their parent assembly.

As C++ programmers would expect, except for the default member accessibility, a ref struct is just like a ref class. We'll refer to both as ref classes.

Every ref class has a base type. If one is not explicitly specified, the default base is `System::Object`. A ref class can have only one base class.

1.1.3 Properties

Regardless of how a Point is represented internally, we think of that Point as having an X and a Y *property*. If the Point actually uses Cartesian representation, the implementation of these properties is trivial. If it uses polar representation, that's more complicated, but it's still a hidden implementation detail.

A *scalar property* is a member that provides field-like access to an instance. For example, in case 3a, we define a property X with type `int`. The token `property` is a contextual keyword, not a globally reserved keyword (although the Visual C++ editor color-codes it as if it were; which is not a bad thing). Its use is only reserved in this context.

A property can have either or both a *get accessor* and a *set accessor*. We'll simply call them the *getter* and the *setter*, respectively. The job of a getter (see case 3b) is to return the value of the given property (by retrieving it from some internal storage, by computing it, or by reading it from a file, for example). The job of a setter (see case 3c) is to set the value of the given property using the programmer-supplied value. These accessors are defined as separate functions with the names `get` and `set`, respectively, and they must return and take, respectively, the declared type of the property, in this case, `int`. (The names `get` and `set` are *not* keywords.) The getter and setter can have different accessibilities; it is quite reasonable to want a public getter and a private or protected setter, for example.

---

[1] While having a keyword that contains whitespace may seem odd, it was done that way to avoid breaking existing code.

5. Value Class Types

In all previous chapters, the class types we've used have been ref classes, which means that instances of them—including those declared on the stack—are managed by the garbage-collector. In this chapter, we'll look at what is often referred to as a "light-weight" class mechanism, namely, the *value class*, instances of which are not managed by the garbage collector.

Value class types are particularly useful for reasonably small data structures that have value semantics. Examples include points in a coordinate system and complex numbers. Typically, a good candidate for implementation as a value class will have only a few data members, will not require inheritance, and will not be expensive in terms of passing and returning by value, or copying during assignment.

5.1 Point as a Value Class

Let's take the `Point` class from §1, and turn it into a value class (see directory vc01\Point):

```
using namespace System;

public value class Point
{
        int x;
        int y;
public:

// define read-write instance properties X and Y

        property int X
        {
                int get() { return x; }
                void set(int val) { x = val; }
        }

        property int Y
        {
                int get() { return y; }
                void set(int val) { y = val; }
        }
```

```
// define instance constructors

        Point(int xor, int yor)
        {
                X = xor;
                Y = yor;
        }

        void Move(int xor, int yor)
        {
                X = xor;
                Y = yor;
        }

        virtual bool Equals(Object^ obj) override
        {
                if (obj == nullptr)
                {
                        return false;
                }
                if (GetType() == obj->GetType())
                {
                        Point^ p = static_cast<Point^>(obj);
                        return (X == p->X) && (Y == p->Y);
                }
                return false;
        }

        static bool operator==(Point p1, Point p2)
        {
                return (p1.X == p2.X) && (p1.Y == p2.Y);
        }

//      static bool operator==(Point% p1, Point% p2)
//      {
//              return (p1.X == p2.X) && (p1.Y == p2.Y);
//      }

//      static bool operator==(Point& p1, Point& p2)
//      {
//              return (p1.X == p2.X) && (p1.Y == p2.Y);
//      }
```

```
        virtual int GetHashCode() override
        {
                return X ^ (Y << 1);
        }

        virtual String^ ToString() override
        {
                return String::Concat("(", X, ",", Y, ")");
        }
};
```

This is achieved by replacing `ref` with `value`. Like `ref class`, `value class` is a keyword containing whitespace. And as we should expect, the only difference between a value class and a value struct, is that the default accessibility for the former is private, while that for the latter is public.

A value class automatically derives from `System::ValueType` (which, in turn, derives from `System::Object`); however, this cannot be declared explicitly. A value class is also implicitly *sealed*; that is, it cannot be used as a base class. (As a result, there is no point giving a value class member an access specifier of `protected`; however, that is not prohibited.) Any value (or ref) class X can be declared sealed explicitly; as follows:

```
value class X sealed {/*…*/};
```

Note the absence of a default constructor, which, in the ref class version, sets the x- and y-coordinates to zero. For a value class, the CLI itself sets all fields in any instance of that class to all-bits-zero; the programmer cannot provide his/her own default constructor. For our `Point` type, this means a default set of coordinates of (0,0), and that is acceptable; however, zero, false, and/or `nullptr` might not be appropriate default values for fields in other types, so this requirement may rule out a value class's being used instead of a ref class for certain types. (Note that a conforming C++/CLI implementation is required to represent the values `false` and `nullptr` as all-bits-zero.)

Another restriction on value classes is that they come with a default copy constructor and assignment operator, both of which perform bitwise copies, and the programmer cannot provide his/her own versions of these functions.

Function `Equals` can be made a bit simpler that the ref class version, but not much. Remember, we are overriding the version defined in `System::Object`, and that takes an `Object^`. Since an argument of that type could have the value `nullptr`, as usual, we deal with first. The step we can omit is that which checks to see if we are being asked to compare something against itself. For ref class implementations of `Equals`, this step is necessary, as an infinite number of handles can refer to the same object. However, in the case of a value class, no two instances can ever represent the same instance. Two instances can represent Points having the exact same coordinates, but changing the x-coordinate of one Point does not change that in the other.

When an instance of `Point` is passed to `Equals`, being of a value class type (which is ultimately derived from `System::Object`), boxing occurs. That is, an instance of `Object` is allocated on the garbage-collected heap, and that instance contains a copy of the Point passed. Since we have just created a new object, and there is only one handle to it, it can never be the same Point object as any other Point object.

10. Destruction and Finalization

C++/CLI is a marriage of two different worlds, one bounded by Standard C++, the other by Standard CLI. Consider the cleanup sometimes needed at the end of an object's life. On the one hand, Standard C++ supports deterministic object cleanup via a *destructor*. On the other, Standard CLI supports non-deterministic object cleanup via a *finalizer*. Since C++/CLI supports both, programmers need to understand both facilities, and to be able to determine which to provide for the classes they develop. They also need to consider that C++/CLI classes might be derived from classes written in other CLI-based languages (such as C# and VB.NET), and vice versa.

In this chapter, we'll look at the differences between destruction and finalization, and the syntax to achieve both.

10.1 The Resource Leakage Problem

Many class types are *self-contained*; that is, an instance of them directly contains all the data needed to represent a value of that type, so no cleanup is needed at the end of the instance's life. However, not all class types are self-contained. For example, a class that implements a variable-length vector typically contains an address to the vector's elements, and a vector length. The vector elements themselves are stored outside the so-called "vector" object, in which case, the vector type is really a descriptor for the data logically stored in that vector. As such, this type needs to have an assignment operator and copy constructor, so instances can be copied correctly. It also needs a destructor to release the vector element space when an instance is destroyed.

If an object acquires some resource or takes some action during its life that needs to be released or undone at the end of its life, in Standard C++, we use a destructor. Here are relevant excerpts from a vector class that exhibits this behavior (see directory df01\Vector_Test):

```
template <typename T>
class Vector
{
        int length;
        T *vector;
public:
        Vector(int vectorLength, T initValue)
        {
                length = vectorLength;
                vector = new T[length];
                // …
        }
```

```
        Vector(const Vector& v)
        {
                length = v.length;
                vector = new T[length];
                // …
        }

        ~Vector()
        {
                delete [] vector;
        }

        T& operator[](int index);
        const T& operator[](int index) const;
        Vector& operator=(const Vector& v);
};
```

The main concern here is that the resource acquired not be leaked. As to exactly when that resource is reclaimed is often of little or no interest, so long as it's done. There are important cases, however, in which we care about the timing of such reclamation. For an example, if an object is backed-up by a file, we probably want that file to be closed immediately that object is destroyed, so the file is available for use by other objects and functions.

Sometimes the life of some resource allocation isn't tied to the life on any one object. In such cases, it is useful to be able to transfer ownership of that release from one object to another. (The standard library type `auto_ptr` is an example of this.) In other situations, it is useful to keep a resource only as long as one or more objects are currently using it, and to free it when it is no longer being used. This is achieved by a technique known as *reference counting*.

We design a Standard C++ class so that instances of it can be allocated statically (either as globals or at file-scope), automatically (on the stack), or dynamically (on the native heap). The order in which constructors and destructors are called is, essentially, well defined and well understood.

10.2 Automatic Garbage Collection

In §1.2.1and §1.2.2, we were introduced to allocating memory on the managed heap, and garbage collection. Let's recap what we learned there by looking at the following example (see directory df02):

```
using namespace System::Text;

StringBuilder^ f(StringBuilder^ sb)
{
        // ...
/*1*/   return gcnew StringBuilder(25);
}
```

```
int main()
{
/*2*/   StringBuilder^ sb1 = gcnew StringBuilder(100);
/*3*/   StringBuilder^ sb2 = f(sb1);
/*4*/   sb1 = nullptr;
}
```

In case 2, we define an automatic handle to type `StringBuilder`, and we make it refer to the new StringBuilder object allocated on the managed heap. This memory is under the watchful eye of the garbage collector.

The way to think about garbage collection is that every distinct object allocated on the managed heap has its own handle reference count, which is maintained by the system, and which is not available—even for read access—to the programmer. When that count reaches zero, that memory can be reclaimed automatically by the system.

When the object is created in case 2, its reference count is 1. When `f` in called in case 3, `sb` now also refers to the same object, making its reference count 2. When `f` terminates, handle `sb` goes out of scope, in which case, the memory to which it referred has one less handle associated with it, making its reference count 1. If a handle doesn't go out of scope for some time, yet we are no longer interested in the memory to which it refers, we can reduce its reference count by setting that handle to `nullptr`, as in case 4. At this time, there are no handles referring to the initial allocated object, make it available for garbage collection.

In case 1, we allocate a second object, so its reference count is 1. Because we return the handle by value, for an instant, two handles refer to that new object; however, one of them goes out of scope when `f` terminates. The returned handle is used to initialize `sb2` in case 3 making the count 2, but the handle returned is no longer used, so the effective count is still only 1. When `main` terminates, `sb2` goes out of scope, the reference count to which it refers is decremented, to zero, so it becomes eligible for garbage collection.

Note that there is no way to explicitly free managed memory, however; we can apply `delete` to a handle, and that will run the destructor for the object immediately, but the memory will not be reclaimed until the garbage collector decides it needs to collect it.

10.3 The Vector Class Revisited

Let us now re-implement the vector class using C++/CLI and garbage collection (see directory df03\Vector):

```
generic <typename T>
public ref class Vector
{
        int length;
        array<T>^ vector;
public:
        property T default[int] { /* … */ }
```