C++/CLI

The Visual C++ Language for .NET

Gordon Hogenson

Apress®

C++/CLI: The Visual C++ Language for .NET

ISBN-13: 978-1-59059-705-7

ISBN-10: 1-59059-705-2

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

Lead Editors: Ewan Buckingham, James Huddleston
Technical Reviewer: Damien Watkins
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Julie M. Smith
Copy Edit Manager: Nicole Flores
Copy Editor: Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Cheu
Compositor: Susan Glinert Stevens
Proofreader: Elizabeth Berry
Indexer: John Collin
Artist: Kinetic Publishing Services, LLC
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 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code/Download section.

CHAPTER 2

A Quick Tour of the C++/CLI Language Features

The aim of this chapter is to give you a general idea of what C++/CLI is all about by providing a brief look at most of the new language features in the context of an extended example, saving the details for later chapters. By the end of this chapter, you'll have a good idea of the scope of the most important changes and will be able to start writing some code.

Primitive Types

The CLI contains a definition of a new type system called the common type system (CTS). It is the task of a .NET language to map its own type system to the CTS. Table 2-1 shows the mapping for C++/CLI.

Table 2-1. *Primitive Types and the Common Type System*

CLI Type	C++/CLI Keyword	Declaration	Description
Boolean	bool	bool isValid = true;	true or false
Byte	unsigned char	unsigned char c = 'a';	8-bit unsigned integer
Char	wchar_t	wchar_t wc = 'a' or L'a';	Unicode character
Double	double	double d = 1.0E-13;	8-byte double-precision floating-point number
Int16	short	short s = 123;	16-bit signed integer
Int32	long, int	int i = -1000000;	32-bit signed integer
Int64	__int64, long long	__int64 i64 = 2000;	64-bit signed integer
SByte	char	char c = 'a';	Signed 8-bit integer
Single	float	float f = 1.04f;	4-byte single-precision floating-point number

Table 2-1. *Primitive Types and the Common Type System (Continued)*

CLI Type	C++/CLI Keyword	Declaration	Description
UInt16	unsigned short	unsigned short s = 15;	Unsigned 16-bit signed integer
UInt32	unsigned long, unsigned int	unsigned int i = 500000;	Unsigned 32-bit signed integer
UInt64	unsigned __int64, unsigned long long	unsigned __int64 i64 = 400;	Unsigned 64-bit integer
Void	void	n/a	Untyped data or no data

In this book, the term *managed type* refers to any of the CLI types mentioned in Table 2-1, or any of the aggregate types (`ref class`, `value class`, etc.) mentioned in the next section.

Aggregate Types

Aggregate types in C++ include structures, unions, classes, and so on. C++/CLI provides managed aggregate types. The CTS supports several kinds of aggregate types:

- `ref class` and `ref struct`, a reference type representing an object
- `value class` and `value struct`, usually a small object representing a value
- `enum class`
- `interface class`, an interface only, with no implementation, inherited by classes and other interfaces
- Managed arrays
- Parameterized types, which are types that contain at least one unspecified type that may be substituted by a real type when the parameterized type is used

Let's explore these concepts together by developing some code to make a simple model of atoms and radioactive decay. First, consider an atom. To start, we'll want to model its position and what type of atom it is. In this initial model, we're going to consider atoms to be like the billiard balls they were once thought to be, before the quantum revolution changed all that. So we will for the moment consider that an atom has a definite position in three-dimensional space. In classic C++, we might create a class like the one in the upcoming listing, choosing to reflect the *atomic number*—the number of protons, which determines what type of element it is; and the *isotope number*—the number of protons plus the number of neutrons, which determines which isotope of the element it is. The isotope number can make a very innocuous or a very explosive difference in practical terms (and in geopolitical terms). For example, you may have heard of carbon dating, in which the amount of radioactive carbon-14 is measured to determine the age of wood or other organic materials. Carbon can have an isotope number of 12, 13, or 14. The most common isotope of carbon is carbon-12, whereas carbon-14 is a radioactive isotope. You may also have heard a lot of controversy about isotopes of uranium.

There's a huge geopolitical difference between uranium-238, which is merely mildly radioactive, and uranium-235, which is the principal ingredient of a nuclear bomb.

In this chapter, together we'll create a program that simulates radioactive decay, with specific reference to carbon-14 decay used in carbon dating. We'll start with a fairly crude example, but by the end of the chapter, we'll make it better using C++/CLI constructs. Radioactive decay is the process by which an atom changes into another type of atom by some kind of alteration in the nucleus. These alterations result in changes that transform the atom into a different element. Carbon-14, for example, undergoes radioactive decay by emitting an electron and changing into nitrogen-14. This type of radioactive decay is referred to as β^- (*beta minus* or simply *beta*) decay, and always results in a neutron turning into a proton in the nucleus, thus increasing the atomic number by 1. Other forms of decay include β^+ (*beta plus* or *positron*) decay, in which a positron is emitted, or alpha decay, in which an alpha particle (two protons and two neutrons) is ejected from the nucleus. Figure 2-1 illustrates beta decay for carbon-14.

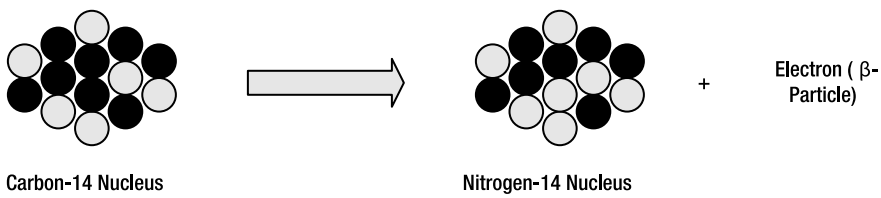

Figure 2-1. *Beta decay. Carbon-14 decays into nitrogen-14 by emitting an electron. Neutrons are shown in black; protons in gray.*

Listing 2-1 shows our native C++ class modeling the atom.

Listing 2-1. *Modeling an Atom in Native C++*

```
// atom.cpp
class Atom
{
   private:
      double pos[3];
      unsigned int atomicNumber;
      unsigned int isotopeNumber;

   public:
       Atom() : atomicNumber(1), isotopeNumber(1)
       {
           // Let's say we most often use hydrogen atoms,
           // so there is a default constructor that assumes you are
           // creating a hydrogen atom.
           pos[0] = 0; pos[1] = 0; pos[2] = 0;
       }
```

```
        Atom(double x, double y, double z, unsigned int a, unsigned int n)
         : atomicNumber(a), isotopeNumber(n)
        {
           pos[0] = x; pos[1] = y; pos[2] = z;
        }
        unsigned int GetAtomicNumber() { return atomicNumber; }
        void SetAtomicNumber(unsigned int a) { atomicNumber = a; }
        unsigned int GetIsotopeNumber() { return isotopeNumber; }
        void SetIsotopeNumber(unsigned int n) { isotopeNumber = n; }
        double GetPosition(int index) { return pos[index]; }
        void SetPosition(int index, double value) { pos[index] = value; }
};
```

You could compile the class unchanged in C++/CLI with the following command line:

```
cl /clr atom.cpp
```

and it would be a valid C++/CLI program. That's because C++/CLI is a superset of C++, so any C++ class or program is a C++/CLI class or program. In C++/CLI, the type in Listing 2-1 (or any type that could have been written in classic C++) is a native type.

Reference Classes

Recall that the managed types use `ref class` (or `value class`, etc.), whereas the native classes just use `class` in the declaration. Reference classes are often informally referred to as *ref classes* or *ref types*. What happens if we just change `class Atom` to `ref class Atom` to see whether that makes it a valid reference type? (The `/LD` option tells the linker to generate a DLL instead of an executable.)

```
C:\ >cl /clr /LD atom1.cpp
atom1.cpp(4) : error C4368: cannot define 'pos' as a member of managed 'Atom':
mixed types are not supported
```

Well, it doesn't work. Looks like there are some things that we cannot use in a managed type. The compiler is telling us that we're trying to use a native type in a reference type, which is not allowed. (In Chapter 12, you'll see how to use interoperability features to allow some mixing.)

I mentioned that there is something called a managed array. Using that instead of the native array should fix the problem, as in Listing 2-2.

Listing 2-2. *Using a Managed Array*

```
// atom_managed.cpp
ref class Atom
{
   private:
      array<double>^ pos;  // Declare the managed array.
      unsigned int atomicNumber;
      unsigned int isotopeNumber;
```

```
   public:
       Atom()
       {
           // We'll need to allocate space for the position values.
           pos = gcnew array<double>(3);
           pos[0] = 0; pos[1] = 0; pos[2] = 0;
           atomicNumber = 1;
           isotopeNumber = 1;
       }
       Atom(double x, double y, double z, unsigned int atNo, unsigned int n)
        : atomicNumber(atNo), isotopeNumber(n)
       {
          // Create the managed array.
          pos = gcnew array<double>(3);
          pos[0] = x; pos[1] = y; pos[2] = z;
       }
       // The rest of the class declaration is unchanged.
};
```

So we have a `ref class Atom` with a managed array, and the rest of the code still works. In the managed type system, the array type is a type inheriting from `Object`, like all types in the CTS. Note the syntax used to declare the array. We use the angle brackets suggestive of a template argument to specify the type of the array. Don't be deceived—it is not a real template type. Notice that we also use the handle symbol, indicating that `pos` is a handle to a type. Also, we use `gcnew` to create the array, specifying the type and the number of elements in the constructor argument instead of using square brackets in the declaration. The managed array is a reference type, so the array and its values are allocated on the managed heap.

So what exactly can you embed as fields in a managed type? You can embed the types in the CTS, including primitive types, since they all have counterparts in the CLI: `double` is `System::Double`, and so on. You cannot use a native array or native subobject. However, there is a way to reference a native class in a managed class, as you'll see in Chapter 12.

Value Classes

You may be wondering if, like the `Hello` type in the previous chapter, you could also have created `Atom` as a value type. If you only change `ref` to `value` and recompile, you get an error message that states "value types cannot define special member functions"—this is because of the definition of the default constructor, which counts as a special member function. Thanks to the compiler, value types always act as if they have a built-in default constructor that initializes the data members to their default values (e.g., zero, `false`, etc.). In reality, there is no constructor emitted, but the fields are initialized to their default values by the CLR. This enables arrays of value types to be created very efficiently, but of course limits their usefulness to situations where a zero value is meaningful.

Let's say you try to satisfy the compiler and remove the default constructor. Now, you've created a problem. If you create an atom using the built-in default constructor, you'll have atoms with atomic number zero, which wouldn't be an atom at all. Arrays of value types don't call the constructor; instead, they make use of the runtime's initialization of the value type

fields to zero, so if you wanted to create arrays of atoms, you would have to initialize them after constructing them. You could certainly add an `Initialize` function to the class to do that, but if some other programmer comes along later and tries to use the atoms before they're initialized, that programmer will get nonsense (see Listing 2-3).

Listing 2-3. *C++/CLI's Version of Heisenberg Uncertainty*

```
void atoms()
{
   int n_atoms = 50;
   array<Atom>^ atoms = gcnew array<Atom>(n_atoms);

   // Between the array creation and initialization,
   // the atoms are in an invalid state.
   // Don't call GetAtomicNumber here!

   for (int i = 0; i < n_atoms; i++)
   {
       atoms[i].Initialize( /* ... */ );
   }
}
```

Depending on how important this particular drawback is to you, you might decide that a value type just won't work. You have to look at the problem and determine whether the features available in a value type are sufficient to model the problem effectively. Listing 2-4 provides an example where a value type definitely makes sense: a `Point` class.

Listing 2-4. *Defining a Value Type for Points in 3D Space*

```
// value_struct.cpp
value struct Point3D
{
   double x;
   double y;
   double z;

};
```

Using this structure instead of the array makes the `Atom` class look like Listing 2-5.

Listing 2-5. *Using a Value Type Instead of an Array*

```
ref class Atom
{
   private:
      Point3D position;
      unsigned int atomicNumber;
      unsigned int isotopeNumber;
```

```
    public:
        Atom(Point3D pos, unsigned int a, unsigned int n)
         : position(pos), atomicNumber(a), isotopeNumber(n)
        { }

        Point3D GetPosition()
        {
           return position;
        }
        void SetPosition(Point3D new_position)
        {
           position = new_position;
        }

        // The rest of the code is unchanged.

};
```

The value type `Point3D` is used as a member, return value, and parameter type. In all cases you use it without the handle. You'll see later that you can have a handle to a value type, but as this code is written, the value type is copied when it is used as a parameter, and when it is returned. Also, when used as a member for the `position` field, it takes up space in the memory layout of the containing `Atom` class, rather than existing in an independent location. This is different from the managed array implementation, in which the elements in the `pos` array were in a separate heap location. Intensive computations with this class using the value struct should be faster than the array implementation. This is the sweet spot for value types—they are very efficient for small objects.

Enumeration Classes

So, you've seen all the managed aggregate types except interface classes and enumeration classes. The enumeration class (or *enum class* for short) is pretty straightforward. It looks a lot like a classic C++ enum, and like the C++ enum, it defines a series of named values. It's actually a value type. Listing 2-6 is an example of an enum class.

Listing 2-6. *Declaring an Enum Class*

```
// elements_enum.cpp

enum class Element
{
   Hydrogen = 1, Helium, Lithium, Beryllium, Boron, Carbon, Nitrogen, Oxygen,
   Fluorine, Neon
   // ... 100 or so other elements omitted for brevity
};
```

While we could have listed these in the order they appear in the Tom Lehrer song "The Elements" (a classic sung to the tune of "Major-General's Song"), we'll list them in order of increasing atomic number, so we can convert between element type and atomic number easily.

The methods on the enum class type allow a bit of extra functionality that you wouldn't get with the old C++ enum. For example, you can call the `ToString` method on the enum and use that to print the named value. This is possible because the enum class type, like all .NET types, derives from `Object`, and `Object` has a `ToString` method. The .NET Framework `Enum` type overrides `ToString`, and that implementation returns the enum named value as a `String`. If you've ever written a tedious switch statement in C or C++ to generate a string for the value of an enum, you'll appreciate this convenience. We could use this `Element` enum in our `Atom` class by adding new method `GetElementType` to the `Atom` class, as shown in Listing 2-7.

Listing 2-7. *Using Enums in the Atom Class*

```
ref class Atom
{
       // ...

       Element GetElementType()
       {
           return safe_cast<Element>( atomicNumber );
       }
       void SetElementType(Element element)
       {
           atomicNumber = safe_cast<unsigned int>(element);
       }
       String^ GetElementString()
       {
          return GetElementType().ToString();
       }
};
```

Notice a few things about this code. Instead of the classic C++ `static_cast` (or `dynamic_cast`), we use a casting construct that is introduced in C++/CLI, `safe_cast`. A *safe cast* is a cast in which there is, if needed, a runtime check for validity. Actually, there is no check to see whether the value fits within the range of defined values for that enum, so in fact this is equivalent to `static_cast`.

Because `safe_cast` is safer for more complicated conversions, it is recommended for general use in code targeting the CLR. However, there may be a performance loss if a type check must be performed at runtime. The compiler will determine whether a type check is actually necessary, so if it's not, the code is just as efficient as with another form of cast. If the type check fails, `safe_cast` throws an exception. Using `dynamic_cast` would also result in a runtime type check, the only difference being that `dynamic_cast` will never throw an exception. In this particular case (Listing 2-7), the compiler knows that the enum value will never fail to be converted to an unsigned integer.

Interface Classes

Interfaces are not something that is available in classic C++, although something like an interface could be created by using an abstract base class in which all the methods are pure virtual (declared with `= 0`), which would mean that they had no implementation. Even so, such a class is not quite the same as an interface. An interface class has no fields and no method implementations; an abstract base class may have these. Also, multiple interface classes may be inherited by a class, whereas only one noninterface class may be inherited by a managed type.

We want to model radioactive decay. Since most atoms are not radioactive, we don't want to add radioactivity methods to our `Atom` class, but we do want another class, maybe `RadioactiveAtom`, which we'll use for the radioactivity modeling. We'll have it inherit from `Atom` and add the extra functionality for radioactive decay. It might be useful to have all the radioactivity methods defined together so we can use them in another class. Who knows, maybe we'll eventually want to have a version of an `Ion` class that also implements the radioactivity methods so we can have radioactive atoms with charge, or something. In classic C++, we might be tempted to use multiple inheritance. We could create a `RadioactiveIon` class that inherits from both `Ion` and `RadioactiveAtom`. But we can't do that in C++/CLI (at least not in a managed type) because in C++/CLI managed types are limited to only one direct base class. However, a class may implement as many interface classes as are needed, so that is a good solution. An interface defines a set of related methods; implementing an interface indicates that the type supports the functionality defined by that interface. Many interfaces in the .NET Framework have names that end in "able," for example, `IComparable`, `IEnumerable`, `ISerializable`, and so on, suggesting that interfaces deal with "abilities" of objects to behave in a certain way. Inheriting from the `IComparable` interface indicates that objects of your type support comparison functionality; inheriting from `IEnumerable` indicates that your type supports iteration via .NET Framework enumerators; and so on.

If you're used to multiple inheritance, you may like it or you may not. I thought it was a cool thing at first, until I tried to write up a complicated type system using multiple inheritance and virtual base classes, and found that as the hierarchy got more complicated, it became difficult to tell which virtual method would be called. I became convinced that the compiler was calling the wrong method, and filed a bug report including a distilled version of my rat's nest inheritance hierarchy. I was less excited about multiple inheritance after that. Whatever your feelings about multiple inheritance in C++, the inheritance rules for C++/CLI types are a bit easier to work with.

Using interfaces, the code in Listing 2-8 shows an implementation of `RadioactiveAtom` that implements the `IRadioactive` interface.

Note the absence of the `public` keyword in the base class and interface list. Inheritance is always public in C++/CLI, so there is no need for the `public` keyword.

Listing 2-8. *Defining and Implementing an Interface*

```
// atom_interfaces.cpp

interface class IRadioactive
{
   void AlphaDecay();
   void BetaDecay();
```

```
   double GetHalfLife();
};

ref class RadioactiveAtom : Atom, IRadioactive
{
       double half_life;

       void UpdateHalfLife()
       {
           // ...
       }


    public:
       // The atom releases an alpha particle
       // so it loses two protons and two neutrons.
       virtual void AlphaDecay()
       {
           SetAtomicNumber(GetAtomicNumber() - 2);
           SetIsotopeNumber(GetIsotopeNumber() - 4);
           UpdateHalfLife();
       }

       // The atom releases an electron.
       // A neutron changes into a proton.
       virtual void BetaDecay()
       {
           SetAtomicNumber(GetAtomicNumber() + 1);
           UpdateHalfLife();
       }

       virtual double GetHalfLife()
       {
           return half_life;
       }
};
```

The plan is to eventually set up a loop representing increasing time, and "roll the dice" at each step to see whether each atom decays. If it does, we want to call the appropriate decay method, either beta decay or alpha decay. These decay methods of the `RadioactiveAtom` class will update the atomic number and isotope number of the atom according to the new isotope that the atom decayed to. At this point, in reality, the atom could still be radioactive, and would then possibly decay further. We would have to update the half-life at this point. In the next sections, we will continue to develop this example.

The previous sections demonstrated the declaration and use of managed aggregate types, including ref classes, value classes, managed arrays, enum classes, and interface classes. In the next section, you'll learn about features that model the "*has-a*" relationship for an object: properties, delegates, and events.

Elements Modeling the "has-a" Relationship

One thing you've probably noticed by now in our `Atom` class is there are a lot of methods that begin with `Get` and `Set` to capture the "has-a" relationship between an object and the properties of the object. Some of the C++/CLI features were added simply to capture such commonly used patterns in the language. Doing this helps standardize common coding practices, which can help in making code more readable. Language features in C++/CLI supporting the "has-a" relationship include *properties* and *events*.

Properties

C++/CLI provides language support for properties. *Properties* are elements of a class that are represented by a value (or set of indexed values for indexed properties). Many objects have properties, and making this a first-class language construct, even if at first they might seem a trivial addition, does make life easier. Let's change all the `Get` and `Set` methods and use properties instead. For simplicity we'll return to the example without the interfaces (see Listing 2-9).

Listing 2-9. *Using Properties*

```
ref class Atom
{
   private:
      array<double>^ pos;

   public:

       Atom(double x, double y, double z, unsigned int a, unsigned int n)
       {
          pos = gcnew array<double>(3);
          pos[0] = x; pos[1] = y; pos[2] = z;

          AtomicNumber = a;
          IsotopeNumber = n;
       }

       property unsigned int AtomicNumber;
       property unsigned int IsotopeNumber;

       property Element ElementType
       {
          Element get()
          {
              return safe_cast<Element>(AtomicNumber);
          }
```

```
            void set(Element element)
            {
                AtomicNumber = safe_cast<int>(element);
            }
        }

        property double Position[int]
        {
            // If index is out of range, the array access will
            // throw an IndexOutOfRange exception.
            double get(int index)            {
               return pos[index];
            }
            void set(int index, double value)
            {
               pos[index] = value;
            }
        }
};
```

We create four properties: `AtomicNumber`, `IsotopeNumber`, `ElementType`, and `Position`. We deliberately use three different ways of defining these properties to illustrate the range of what you can do with properties. The `ElementType` property is the standard, commonly used form. The property is named, followed by a block containing the `get` and `set` methods, fully prototyped and implemented. The names of the accessors must be `get` and `set`, although you don't have to implement both. If you implement only one of them, the property becomes read-only or write-only. The `AtomicNumber` and `IsotopeNumber` properties are what's known as trivial properties. *Trivial properties* have getter and setter methods created automatically for them: also notice that we remove the `atomicNumber` and `isotopeNumber` fields. They are no longer needed since private fields are created automatically for trivial properties. The third type of property is known as an *indexed property* or a *vector property*. Nonindexed properties are known as *scalar properties*. The indexed property `Position` is implemented with what looks like array indexing syntax. Vector properties take a value in square brackets and use that value as an index to determine what value is returned.

Also notice that we use the property names just like fields in the rest of the body of the class. This is what makes properties so convenient. In assignment expressions, property `get` and `set` methods are called implicitly as appropriate when a property is accessed or is assigned to.

```
AtomicNumber = safe_cast<int>(element); // set called implicitly
```

You can also use the compound assignment operator (`+=`, `-=`, etc.) and the postfix or prefix operators with properties to simplify the syntax in some cases. For example, consider the `AlphaDecay` method in the `RadioactiveAtom` class. It could be written as shown in Listing 2-10.

Listing 2-10. *Using Compound Assignment Operators with Properties*

```
virtual void AlphaDecay()
{
    AtomicNumber -= 2;
    IsotopeNumber -= 4;
    UpdateHalfLife();
}
```

Delegates and Events

Managed types may have additional constructs for events. Events are based on *delegates*, managed types that are like souped-up function pointers. Delegates are actually more than just a function pointer, because they may refer to a whole set of methods, rather than just one. Also, when referencing a nonstatic method, they reference both the object and the method to call on that object. As you'll see, the syntax, both for declaring a delegate and invoking one, is simpler than the corresponding syntax for using a function pointer or pointer to member. Continuing with the radioactivity problem, we'll now use delegates to implement radioactive decay. Atoms have a certain probability for decay. The probability for decaying during a specific interval of time can be determined from the half-life using the formula

probability of decay = ln 2 / halflife * timestep

The constant λ, known as the decay constant, is used to represent *ln 2 / halflife*, so this could also be just

probability of decay = λ* timestep

Listing 2-11 demonstrates creating a delegate type, in this case `DecayProcessFunc`. The `RadioactiveAtoms` class has a property named `DecayProcess`, which is of that delegate type. This property can be set to the beta decay function (here beta minus decay), the alpha decay function, or perhaps some other rare type of radioactive decay.

The delegate indicates both the object and the method. This is the main difference between a delegate and a pointer to member function in classic C++. Listing 2-11 provides the full code, with the delegate code highlighted. I've removed the interface that was used in Listing 2-8, as it is not central to the discussion now.

Listing 2-11. *Using a Delegate*

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

// This declares a delegate type that takes no parameters.
delegate void DecayProcessFunc();

enum class Element; // same as before
ref class Atom; // same as before, but without the position data
```

```
ref class RadioactiveAtom : Atom
{

    public:
       RadioactiveAtom(int a, int n, bool is_stable, double half_life)
           : Atom(a, n)
       {
            IsStable = is_stable;
            HalfLife = half_life;
            Lambda = Math::Log(2) / half_life;
        }

       // The atom releases an alpha particle
       // so it loses two protons and two neutrons.
       virtual void AlphaDecay()
       {
           AtomicNumber -= 2;
           IsotopeNumber -= 4;
           Update();
       }

       // The atom releases an electron.
       void BetaDecay()
       {
           AtomicNumber++;
           Update();
       }

       property bool IsStable;
       property double HalfLife;
       property double Lambda;
       void Update()
       {
          // In this case we assume it decays to a stable nucleus.
          // nullptr is the C++/CLI way to refer to an unassigned handle.
          DecayProcess = nullptr;
          IsStable = true;
       }

       // Declare the delegate property. We'll call this when
       // an atom decays.
       property DecayProcessFunc^ DecayProcess;

}; // ref class RadioactiveAtom
```

```
void SimulateDecay(int a, int n, double halflife, int step,
                   int max_time, int num_atoms, int seed)
{
   array<RadioactiveAtom^>^ atoms = gcnew array<RadioactiveAtom^>(num_atoms);

   // Initialize the array.
   // We cannot use a for each statement here because the for each
   // statement is not allowed to modify the atoms array.
   for (int i = 0; i < num_atoms; i++)
   {
      atoms[i] = gcnew RadioactiveAtom(a, n, false, halflife);
      // Create the delegate.
      atoms[i]->DecayProcess =
          gcnew DecayProcessFunc(atoms[i], &RadioactiveAtom::BetaDecay);
   }

   Random^ rand = gcnew Random(seed);
   for (int t = 0; t < max_time; t++)
   {
       for each (RadioactiveAtom^ atom in atoms)
       {
            if ((!atom->IsStable) && atom->Lambda * step > rand->NextDouble())
            {
                // Invoke the delegate.
                atom->DecayProcess->Invoke();
            }
       }
   }
}

int main()
{
   // Carbon-14.  Atomic Number: 6  Isotope Number 14
   // Half-Life 5730 years
   // Number of atoms 10000
   // Maximum time 10000
   // Random number seed 7757
   SimulateDecay(6, 14, 5730, 1, 10000, 10000, 7757);
}
```

The delegate code consists of a delegate declaration, indicating what arguments and return types the delegated functions may have. Then, there is the point at which the delegate is created. A delegate is a reference type, so you refer to it using a handle, and you use `gcnew` to create the delegate. If the delegate is going to reference a nonstatic member function, call the delegate constructor that takes both an object pointer and the method to be called, using the address-of operator (&) and the qualified method name. If you're assigning the delegate to a static method, omit the object and just pass the second parameter, indicating the method, like this:

```
atoms[i]->DecayProcess =
          gcnew DecayProcessFunc(&RadioactiveAtom::SomeStaticMethod);
```

So far we've used delegates but not events. An *event* is an abstraction representing something happening. Methods called *event handlers* may be hooked up to events to respond in some way to the event. Events are of type delegate, but as you've seen, delegates themselves may be used independently of events. The delegate forms a link between the source of the event (possibly a user action, or some action initiated by other code) and the object and function that responds in some way to the action. In this case, the `RadioactiveAtom` class will have a `Decay` event, declared as in Listing 2-12.

Listing 2-12. *Declaring an Event*

```
ref class RadioactiveAtom
{
   // other code...

   // the event declaration
   event DecayProcessFunc^ Decay;

};
```

Instead of invoking the delegate directly, we call the event in the client code using function-call syntax. The code that hooks up the event looks the same as with the delegate property (see Listing 2-13).

Listing 2-13. *Hooking Up and Firing an Event*

```
// Hook up the event.
atoms[i]->Decay +=
          gcnew DecayProcessFunc(atoms[i], &RadioactiveAtom::BetaDecay);

// ...

// Fire the event.
a->Decay();
```

It is possible for an event to trigger multiple actions. You'll learn about such possibilities in Chapter 7.

You could certainly refine the design further. Perhaps you are bothered by the fact that every instance of `RadioactiveAtom` contains its own `halflife` and `lambda` properties. You might instead create a static data structure to store the half-life information for every type of isotope. What would this structure look like? It would require two indices to look up: both the atomic number and the isotope. However, a two-dimensional array would be a huge waste of space, since most of the cells would never be used. You might try implementing an isotope table as a *sparse array*—a data structure that can be used like an array but is a hashtable underneath so as to avoid storing space for unused elements. The implementation of such a collection type would probably be a template in classic C++. In C++/CLI, it could be a template or it could be another type of parameterized type, a generic type, which the next section describes.

Generics

While templates are supported in C++/CLI for both native and managed types, another kind of parameterized type has been defined in C++/CLI: *generics*. Generics fulfill a different purpose, providing runtime parameterization of types, whereas templates provide compile-time parameterization of types. You'll explore the implications of this difference in Chapter 11. The .NET Framework 2.0 supports generics and provides generic collection classes. One such class is the generic `List`, which is a dynamic array class that automatically expands to accommodate larger numbers of elements. The `List` class definition would look something like the code in Listing 2-14.

Listing 2-14. *Defining a Generic List Class*

```
generic <typename T>
ref class List
{
   public:
      T Add(T t) { /* ... */ }
      void Remove(T t) { /* ... */ }
      // other methods
};
```

This declaration indicates that `List` is a generic type with one type parameter, `T`. Returning to our example of treating isotopes of the chemical elements, the `List` class is a good choice to represent the isotopes of an element, since each element has a different number of isotopes. The generic `List` collection is exposed as a property in this class. When the `List` object is declared, an actual type (a handle to `Isotope`) is used as the parameter. Handles, rather than direct reference types, are allowed as type parameter substitutions. You can also use a value type without a handle. Listing 2-15 shows an `ElementType` class with the `Isotopes` property, which is a list of isotopes for a particular element.

Listing 2-15. *A Reference Class That Uses the Generic List As a Property*

```
ref class Isotope; // implementation omitted for brevity

ref class ElementType
{

   // other properties specifying the element name, atomic number, etc.

   property List<Isotope^>^ Isotopes;
};
```

Using this generic type is as easy as using the managed array type. The code in Listing 2-16 uses a `for each` statement to iterate through the generic `List` collection to look up an isotope by its number. Assume an `Isotope` class with an `IsotopeNumber` property.

Listing 2-16. *Iterating Through a Generic Collection*

```
ref class ElementType
{
      // omitting other members of the class

      // Find an isotope by number. If not found, return a
      // null handle (nullptr).
      Isotope^ FindIsotope(int isotope_number)
      {
         for each (Isotope^ isotope in Isotopes)
         {
            if (isotope->IsotopeNumber == isotope_number)
               return isotope;
         }
         return nullptr;
      }
};
```

A more complete discussion of generics and managed templates is the subject of Chapter 11. In the meantime, we will use the generic `List` class in examples throughout this book. You'll see in Chapter 11 how to define generic classes and functions.

Summary

In this chapter, you learned some of the important language constructs of C++/CLI. Of course, there are other significant features, and there is much more to say about each feature. You had a quick look at primitive types, various aggregate types, managed arrays, properties, delegates, events, and parameterized types. Later chapters will return to each of these aspects of the language in more detail.

Before doing that, though, let's look at compiling and building C++/CLI programs.