ГЛАВА 2

Быстрое знакомство с языковыми средствами C++/CLI

Цель этой главы состоит в том, чтобы дать общее представление о том, как выглядит язык C++/CLI, бросив беглый взгляд на большинство новых языковых средств в контексте усовершенствования примера программы, откладывая детальное изучение этих средств на более поздние главы. К концу этой главы вы будете хорошо представлять, в чем состоят самые важные изменения языка и будете в состоянии начать писать несложный код.

Примитивные типы

В CLI определена новая система типов, называемая общей системой типов (Common Type System — CTS). Задача языка .NET состоит в том, чтобы отобразить его собственную систему типов на CTS. В табл. 2.1 показано такое отображение для C++/CLI.

В этой книге термин *управляемый тип* относится к любому из типов CLI, упомянутому в табл. 2.1, или любому из составных типов (`ref class` (ссылочный класс), `value class` (класс значений) и т.д.), упомянутому в следующем разделе.

Составные типы

Составные типы в C++ включают структуры, объединения, классы и т.д. C++/CLI имеет также составные управляемые типы. CTS поддерживает несколько видов составных типов:

- `ref class` (ссылочный класс) и `ref struct` (ссылочный тип), представляющий объект;
- `value class` (класс значений) и `value struct`, обычно небольшой объект, представляющий значение;
- `enum class` (класс перечисления);
- `interface class` (класс интерфейса), т.е. только интерфейс без реализации, унаследованной классами и другими интерфейсами;

- управляемые массивы;
- параметризованные типы, которые являются типами, содержащими по крайней мере один неопределенный тип, вместо которого можно подставить (или заменить его) фактическим типом-параметром.

Таблица 2.1. Примитивные типы и общая система типов

Тип CLI	Зарезервированное слово C++/CLI	Объявление	Описание
`Boolean` (Булев)	`bool`	`bool isValid = true;` (`bool isValid =` истина;)	`true` (истина) или `false` (ложь)
`Byte` (Байт)	`unsigned char` (символ без знака)	`unsigned char c = 'a';` (символ без знака `c ='a';`)	8-разрядное целое число без знака)
`Char` (Символ)	`wchar_t`	`wchar_t wc = 'a'` или `L'a';`	Символ уникода
`Double` (Двойная точность)	`double` (двойная точность)	`double d = 1.0E-13;` (двойная точность `d = 1.0E-13;`)	8-байтовое число двойной точности с плавающей точкой
`Int16`	`short` (короткое)	`short s = 123;` (короткое `s = 123;`)	16-разрядное целое число со знаком
`Int32`	`long, int` (длинное, int)	`int i = -1000000;`	32-разрядное целое число со знаком
`Int64`	`__int64`, `long long` (`__int64`, длинное длинное)	`__int64 i64 = 2000;`	64-битовое целое число со знаком
`SByte`	`char` (символ)	`char c = 'a';` (символ `c ='a';`)	Со знаком 8-битовое целое число
`Single` (Одинарная точность)	`float` (с плавающей точкой)	`float f = 1.04f;` (`f` с плавающей точкой `= 1.04f;`)	4-байтовое число с плавающей запятой с одинарной точностью
`UInt16`	`unsigned short` (короткое без знака)	`unsigned short s = 15;` (короткое без знака `s = 15;`)	16-разрядное целое число со знаком или без знака
`UInt32`	`unsigned long` (длинное без знака), `unsigned int` (`int` без знака)	`unsigned int i = 500000;` (`int` без `знака i = 500000;`)	32-разрядное целое число со знаком или без знака
`UInt64`	`unsigned __int64` (без знака `int64`), `unsigned long long` (без знака длинное длинное)	`unsigned __int64 i64 = 400;` (без знака `int64 i64 = 400;`)	64-битовое целое число без знака
`Void` (Пусто)	`void` (пусто)	неприменимо	Данные без контроля типов или данные отсутствуют

Давайте освоим эти понятия (концепции), разрабатывая некоторый код, который позволит построить простую модель атомов и радиоактивного распада. Сначала рассмотрим атом. Для начала мы смоделируем его позицию и укажем, какой это

атом. В этой первоначальной модели мы полагаем, что атомы походят на бильярдные шары, как когда-то думали, когда еще квантовая революция не опровергла все это. Таким образом, мы будем в настоящий момент полагать, что атом имеет определенное положение в трехмерном пространстве. В классическом C++ мы могли бы создать класс, такой как приведенный в следующем листинге, решив отображать атомный номер — количество протонов, которое определяет, какой это элемент, и массовое число — количество протонов плюс количество нейтронов, который определяет, какой это изотоп элемента. Массовое число как раз и определяет, данный элемент совершенно безвредный или очень взрывоопасный на практике (и в геополитическом смысле). Например, вы, возможно, слышали о датировке по радиоуглеродному методу, в котором измеряется количество радиоактивного изотопа углерода с атомным весом 14, для определения возраста древесины или других органических материалов. Углерод может иметь массовое число 12, 13 или 14. Самый распространенный изотоп углерода — углерод-12, тогда как углерод-14 — радиоактивный изотоп. Вы, возможно, также слышали о больших спорах из-за изотопов урана. Есть огромное геополитическое различие между ураном-238, который просто умеренно радиоактивен, и ураном-235, главным компонентом термоядерной бомбы.

В этой главе мы вместе создадим программу, которая моделирует радиоактивный распад углерода-14, который используется в датировке по радиоуглероду. Мы начнем с довольно сырого образца, но к концу главы мы улучшим программу, используя конструкции C++/CLI. Радиоактивный распад — процесс, при котором атом превращается в другой тип атома благодаря некоторым изменениям в ядре. Эти изменения приводят к изменениям, которые преобразовывают атом одного элемента в атом другого элемента. Углерод-14, например, подвергаясь радиоактивному распаду, испускает электрон и превращается в азот-14. Этот тип радиоактивного распада называется β^--распадом (или *бета минус*-распадом или просто *бета-распадом*) и всегда приводит к тому, что нейтрон превращается в протон в ядре, таким образом увеличивая атомный номер на 1. Другие формы распада включают β^+-распад (распад *бета с плюсом* или *позитронный* распад), при котором испускается позитрон, а также альфа-распад, при котором альфа-частица (два протона и два нейтрона) вылетает из ядра. На рис. 2.1 поясняется бета-распад для углерода-14.

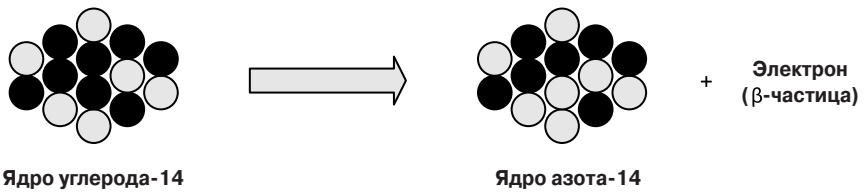

Рис. 2.1. Бета-распад. Углерод-14 после распада превращается в азот-14, испуская электрон. Нейтроны закрашены черным; протоны — серым

В листинге 2.1 показан родной класс C++, моделирующий атом.

Листинг 2.1. Моделирование атома на родном C++

```
// atom.cpp
class Atom                  // класс Атом
{
private:                    // приватный:
```

```
    double pos[3];          // двойная точность;
    unsigned int atomicNumber; // int без знака atomicNumber;
    unsigned int isotopeNumber; // int без знака isotopeNumber;

public: // общедоступный:
    Atom() : atomicNumber(1), isotopeNumber(1) // Атом
    {
        // Поскольку чаще всего встречается атом водорода,
        // то конструктор по умолчанию предполагает
        // создание атома водорода.
        pos[0] = 0; pos[1] = 0; pos[2] = 0;
    }

    Atom(double x, double y, double z, unsigned int a, unsigned int n)
        // Атом(двойная точность x, двойная точность y,
        // двойная точность z, int без знака a, int без знака n),
        : atomicNumber(a), isotopeNumber(n)
    {
        pos[0] = x; pos[1] = y; pos[2] = z;
    }
    unsigned int GetAtomicNumber() { return atomicNumber; }
    // int без знака GetAtomicNumber() {возвращает atomicNumber;}
    void SetAtomicNumber(unsigned int a) { atomicNumber = a; }
    // пусто SetAtomicNumber (int без знака a) {atomicNumber = a;}
    unsigned int GetIsotopeNumber() { return isotopeNumber; }
    // int без знака GetIsotopeNumber() {возвращает isotopeNumber;}
    void SetIsotopeNumber(unsigned int n) { isotopeNumber = n; }
    // пусто SetIsotopeNumber(int без знака n) {isotopeNumber = n;}
    double GetPosition(int index) { return pos[index]; }
    // двойная точность GetPosition(int индекс)
    // {возвращает pos[индекс];}
    void SetPosition(int index, double value) { pos[index] = value; }
    // пусто SetPosition(int индекс, двойная точность значение)
    // { pos[индекс] = значение;}
};
```

Чтобы скомпилировать класс в C++/CLI, ничего в нем не изменяя, можно воспользоваться следующей командной строкой:

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

Это правильная (допустимая) программа на C++/CLI. Поскольку C++/CLI — надмножество C++, то любой класс или программа на C++ является классом или программой на C++/CLI. С точки зрения C++/CLI тип в листинге 2.1 (или любой тип, который допустим в классическом C++) является родным типом.

Ссылочные классы

Давайте вспомним, что в случае управляемых типов используется `ref class` (ссылочный класс) (или `value class` (класс значений) и т.д.), тогда как в объявлении родных классов указывается только слово `class` (класс). Ссылочные классы

часто неофициально называются также классами ссылок или ссылочными типами. Что случится, если мы просто изменим `class Atom` (класс Атом) на `ref class Atom` (ссылочный класс Атом)? Превратится ли он в правильный (допустимый) ссылочный тип? (Опция /LD указывает, что компоновщик должен генерировать DLL вместо исполнимого файла.)

```
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[1]
```

Ладно, это не работает. Похоже, что некоторые вещи нельзя использовать в управляемом типе. Компилятор говорит, что мы используем родной тип в ссылочном типе, а это не позволяется. (В главе 12 "Функциональная совместимость" вы научитесь использовать средства совместимости так, чтобы позволить некоторое смешение.)

Как я уже упоминал, что есть нечто, называемое управляемым массивом. Если использовать его вместо родного массива, программа будет исправлена; она представлена в листинге 2.2.

Листинг 2.2. Использование управляемого массива

```
// atom_managed.cpp
ref class Atom              // ссылочный класс Атом
{
private:                   // приватный:
   array<double>^ pos;      // Объявление управляемого массива.
   // массив<двойная точность>^ pos;
   unsigned int atomicNumber; // int без знака atomicNumber;
   unsigned int isotopeNumber; // int без знака isotopeNumber;

public:                    // общедоступный:
   Atom()                  // Атом ()
   {
      // Мы будем выделять место для значений позиции.
      pos = gcnew array<double>(3);
      // pos = gcnew массив<двойная точность>(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)
      // Атом (двойная точность x, двойная точность y,
      // двойная точность z, int без знака atNo, int без знака n),
      : atomicNumber(atNo), isotopeNumber(n)
   {
      // Создать управляемый массив.
      pos = gcnew array<double>(3);
```

[1] Вот как это можно перевести:
atom1.cpp (4): ошибка C4368: нельзя определять 'pos' как член управляемого 'Атома': смешанные типы не поддерживаются — *Примеч. ред.*

```
        // pos = gcnew массив <двойная точность> (3);
        pos[0] = x; pos[1] = y; pos[2] = z;
    }
    // Остальная часть объявления класса неизменна.
};
```

Таким образом, мы имеем ссылочный класс Атом (`ref class Atom`) с управляемым массивом, причем остальная часть кода работает. В системе управляемых типов тип массива — тип, наследующий `Object` (Объект), как и все типы CTS. Обратите внимание на синтаксис объявления массива. Угловые скобки наводят на размышления о параметре шаблона для определения типа массива. Но не обманывайте себя — на самом деле это не шаблонный тип. Обратите внимание, что символ дескриптора, указывает, что `pos` является дескриптором (маркером) типа. Кроме того, чтобы создать массив, мы используем `gcnew`, определяя тип и количество элементов в параметре конструктора вместо того, чтобы использовать квадратные скобки в объявлении. Управляемый массив — это ссылочный тип, и, таким образом, память для массива и его значений выделяется в управляемой куче.

Итак, что же именно можно вложить в качестве полей в управляемый тип? Типы можно внедрить в CTS, включая и примитивные типы, так как они все имеют своих двойников в CLI: `double` (двойная точность) — `System::Double` (Система:: Двойная точность) и т.д. Тем не менее вы не можете использовать родной массив или родной подобъект. Однако есть способ сослаться на родной класс в управляемом классе, как вы увидите в главе 12 “Функциональная совместимость”.

Классы значений

Вы можете задаться вопросом, можно ли, подобно типу `Hello` (Привет) в предыдущей главе, таким же образом создать `Atom` как тип значений. Если просто заменить `ref` на `value` и перекомпилировать, появится сообщение об ошибках, которое означает, что “типы значений нельзя определять в специальных функциях-членах” — это произойдет из-за определения конструктора по умолчанию, который рассматривается как специальная функция-член. Благодаря компилятору типы значений всегда действуют так, как будто они имеют встроенный конструктор значения по умолчанию, который инициализирует компоненты данных их значениями по умолчанию (например, нулем, ложью и т.д.). В действительности конструктора нет, но поля инициализирует их значениями по умолчанию среда CLR. Это дает возможность очень эффективно создавать массивы типов значений, но, конечно, ограничивает их полноценность теми ситуациями, в которых нужно нулевое значение.

Давайте попробуем удовлетворить компилятор и удалить конструктор по умолчанию. Теперь возникла проблема. Если создать атом, используя встроенный конструктор по умолчанию, то получится атом с нулевым атомным номером, который не является атомом вообще. Массивы типов значений не вызывают конструктор; взамен этого для обнуления они используют инициализацию во время выполнения полей типа значений, так что если нужно создать массивы атомов, то после их построения следует инициализировать их. Чтобы сделать это, можно, конечно, прибавить функцию `Initialize` к классу, но если другой программист впоследс-

твии попробует использовать атомы прежде, чем они будут инициализированы, то получится ерунда (листинг 2.3).

Листинг 2.3. Версия неопределенности Гейзенберга в C++/CLI

```
void atoms()                // пусто атомы()
{
    int n_atoms = 50;
    array<Atom>^ atoms = gcnew array<Atom>(n_atoms);
    // массив <Атом>^ атомы = gcnew массив<Атом>(n_atoms);

    // Между созданием массива и инициализацией
    // атомы находятся в недопустимом состоянии.
    // Не вызывать GetAtomicNumber здесь!

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

В зависимости от того, какое значение для вас имеет этот конкретный недостаток, вы можете решить, что тип значений просто не годится. Следует проанализировать задачу и определить, достаточно ли эффективны функции, доступные для типа значений, чтобы фактически построить модель задачи. В листинге 2.4 приведен пример, в котором тип значений определенно имеет смысл: класс `Point` (Точка).

Листинг 2.4. Определение типа значений для точек в трехмерном пространстве

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

};
```

Если использовать такую структуру вместо массива, то класс `Atom` будет примерно таким, как в листинге 2.5.

Листинг 2.5. Использование типа значений вместо массива

```
ref class Atom              // ссылочный класс Атом
{
private:                    // приватный:
    Point3D position;
    unsigned int atomicNumber; // int без знака atomicNumber;
    unsigned int isotopeNumber; // int без знака isotopeNumber;

public:                     // общедоступный:
    Atom(Point3D pos, unsigned int a, unsigned int n)
```

```
        // Атом (Point3D pos, int без знака a, int без знака n)
        : position(pos), atomicNumber(a), isotopeNumber(n)
    { }

    Point3D GetPosition()
    {
        return position;      // возвратить позицию;
    }
    void SetPosition(Point3D new_position) // пусто
    {
        position = new_position; // позиция
    }

    // Остальная часть кода без изменений.

};
```

Тип значений `Point3D` используется как член, возвращаемое значение и тип параметра. Во всех случаях оно используется без дескриптора. Вы позже увидите, что можно иметь дескриптор для типа значения, но так как написан этот код, тип значений копируется, когда он используется в качестве параметра и когда он возвращается. Кроме того, когда он используется как член для поля `position` (позиция), он занимает место в той памяти, где хранится класс `Atom`, а не существует независимо где-нибудь отдельно. Это отличается от реализации управляемого массива, в котором элементы в массиве `pos` были в отдельном месте кучи. Интенсивные вычисления с этим классом, использующим `value struct`, должны быть более быстрыми, чем при реализации с помощью массива. Это приятная особенность типов значений — они очень эффективны для небольших объектов.

Классы перечислений

Итак, вы уже видели все составные управляемые типы, кроме классов интерфейса и классов перечислений. Класс перечисления (или просто *класс enum* для краткости) является довольно простым. Он во многом походит на классическое перечисление из C++, и, как и перечисление в C++, он определяет ряд именованных значений. Фактически это тип значений. Листинг 2.6 — пример класса перечисления.

Листинг 2.6. Объявление класса перечисления

```
// elements_enum.cpp

enum class Element // класс перечисления Элемент
{
    Hydrogen = 1, Helium, Lithium, Beryllium, Boron, Carbon, Nitrogen,
Oxygen,
    // Водород = 1, Гелий, Литий, Бериллий, Бор, Углерод, Азот, Кислород,
    Fluorine, Neon
    // Фтор, Неон
    // ... Приблизительно 100 других элементов опущены для краткости
};
```

Хотя мы могли бы перечислить химические элементы в том порядке, в каком они появляются в песне Тома Лехрера *Elements* ("Элементы") (классик, напевавший мелодию *Major-General's Song* ("Песня генерал-майора")), мы перечислили их в порядке увеличения атомного номера, благодаря чему мы можем легко преобразовать тип элемента в атомный номер.

Методы класса перечисления имеют немного больше дополнительных средств, чем перечисления в старом C++. Например, вы можете вызвать метод `ToString` для перечисления и использовать для печати именованное значение. Это возможно, потому что тип класса перечисления, как и все типы .NET, является производным от `Object` (Объект), а `Object` имеет метод `ToString`. Тип `Enum` среды .NET Framework переопределяет `ToString`, и полученная реализация возвращает поименованное значение перечисления в виде строки `String`. Если вы когда-либо писали утомительный оператор переключателя `switch` в C или C++, чтобы генерировать строку для значения перечисления, вы оцените это удобство. Это перечисление элементов `Element` можно использовать в нашем классе `Atom`, добавив новый метод `GetElementType` к классу `Atom`, как показано в листинге 2.7.

Листинг 2.7. Использование перечислений в классе **`Atom`**

```
ref class Atom              // ссылочный класс Atom
{
    // ...

    Element GetElementType()  // Элемент GetElementType ()
    {
        return safe_cast<Element>( atomicNumber );
        // возвратить safe_cast<Элемент> (atomicNumber);
    }
    void SetElementType(Element element)
        // пусто SetElementType(Элемент элемент)
    {
        atomicNumber = safe_cast<unsigned int>(element);
        // atomicNumber = safe_cast<int без знака>(элемент);
    }
    String^ GetElementString() // Строка^ GetElementString()
    {
        return GetElementType().ToString(); // возвратить;
    }
};
```

Обратите внимание на несколько моментов в этом коде. Вместо `static_cast` (или `dynamic_cast`) в классическом C++ мы используем конструкцию приведения, которая представлена в C++/CLI как `safe_cast`. Безопасное приведение — приведение, в котором есть, если нужно, динамическая проверка правильности. Фактически не нужно выяснять, попадает ли значение в диапазон (область изменения) определенных значений для этого перечисления, так что фактически это эквивалентно `static_cast`.

Поскольку приведение `safe_cast` безопаснее для более сложных преобразований, именно оно рекомендуется для общего использования в коде, предназначенном для CLR. Однако случается ухудшение рабочих характеристик, если контроль

соответствия типов придется выполнять во время выполнения. Компилятор определит, необходим ли контроль соответствия типов на самом деле, так что если он не нужен, то код столь же эффективен, как и для других форм приведения. Если контроль соответствия типов потерпит неудачу, `safe_cast` вызовет исключение. Если бы мы использовали `dynamic_cast`, то это также привело бы к контролю соответствия типов во время выполнения; единственная разница состоит в том, что `dynamic_cast` никогда не вызывает исключение. В данном конкретном случае (см. листинг 2.7), компилятор знает, что никогда не может возникнуть ситуация, в которой значение перечисления нельзя преобразовать в целое число без знака.

Классы интерфейса

Строго говоря, интерфейсы как таковые недоступны в классическом C++, хотя нечто, используемое в качестве интерфейса, можно создать с помощью абстрактного класса, в котором все методы являются чисто виртуальными (в объявлении применена конструкция `= 0`), что означает, что у них нет реализации. Но даже в этом случае такой класс — совсем не то же самое, что интерфейс. Класс интерфейса не имеет полей и методов реализации; абстрактный же класс может иметь их. Кроме того, класс может унаследовать несколько классов интерфейса, тогда как управляемым типом может быть унаследован только один неинтерфейсный класс.

Мы хотим построить модель радиоактивного распада. Так как большинство атомов не радиоактивно, мы не будем добавлять методы радиоактивности к нашему классу `Atom`, но мы действительно хотим другой класс, возможно назовем его `RadioactiveAtom`, который мы будем использовать для моделирования радиоактивности. Мы сделаем так, чтобы он был наследником класса `Atom`, и добавим дополнительные функциональные возможности для моделирования радиоактивного распада. Было бы полезно все методы радиоактивности определить вместе, чтобы можно было использовать их в другом классе. Кто знает, возможно мы в конечном счете захотим иметь версию класса `Ion` (Ион), который также реализует методы радиоактивности, таким образом мы можем иметь радиоактивные атомы с зарядом или еще с кое-чем другим. В классическом C++ мы могли бы попытаться использовать множественное наследование. Мы могли бы создать класс `RadioactiveIon`, который является наследником и `Ion`, и `RadioactiveAtom`. Но мы не можем сделать этого в C++/CLI (по крайней мере не можем в управляемом типе), потому что в C++/CLI управляемые типы могут иметь только один (непосредственный) базовый класс. Однако в классе можно реализовать столько классов интерфейса, сколько нужно для хорошего решения. В интерфейсе определяется множество связанных методов; реализация интерфейса указывает, что тип поддерживает функциональные возможности, определенные в данном интерфейсе. Многие интерфейсы в .NET Framework имеют имена, которые заканчиваются на *-able*, например, `IComparable`, `IEnumerable`, `ISerializable` и т.д., что предлагает, что интерфейсы имеют дело со “способностями” (abilities) объектов вести себя определенным способом. Наследование интерфейса `IComparable` указывает, что объекты данного типа поддерживают функциональные возможности сравнения; наследование `IEnumerable` указывает, что данный тип поддерживает итерацию с помощью перечислителей .NET Framework, и т.д.

Если вы привыкли к множественному наследованию, вы можете любить или не любить его. Сначала я думал, что это была "крутая" вещь, пока я не попробовал написать сложную систему типов, используя множественное наследование и виртуальные базовые классы. В результате я увидел, что, по мере того как иерархия получалась все более сложной, стало трудно сказать, какой виртуальный метод вызовется. Я был убежден, что компилятор вызывал не тот метод, и одно за другим регистрировал сообщения об ошибках, все время упрощая версию запутанной иерархии наследственности моей программы. После этого мое восхищение от множественного наследования поохладело. Безотносительно ваших чувств о множественном наследовании в C++ с правилами наследственности для типов в C++/CLI работать немного легче.

Интерфейсы используются в листинге 2.8, в котором показан код реализации `RadioactiveAtom`, которая реализует интерфейс `IRadioactive`.

Заметьте, что зарезервированное слово `public` (общедоступный) в базовом классе и списке интерфейса отсутствует. Наследственность всегда общедоступна (`public`) в C++/CLI, и потому нет потребности в зарезервированном слове `public`.

Листинг 2.8. Определение и реализация интерфейса

```
// atom_interfaces.cpp

interface class IRadioactive  // класс интерфейса IRadioactive
{
   void AlphaDecay();       // пусто AlphaDecay();
   void BetaDecay();        // пусто BetaDecay();
   double GetHalfLife();    // двойная точность GetHalfLife ();
};

ref class RadioactiveAtom : Atom, IRadioactive
// ссылочный класс RadioactiveAtom: Атом, IRadioactive
{
   double half_life;        // двойная точность half_life;

   void UpdateHalfLife()    // пусто UpdateHalfLife()
   {
      // ...
   }


public:                     // общедоступный:
   // Атом выбрасывает альфа-частицу,
   // таким образом, он теряет два протона и два нейтрона.
   virtual void AlphaDecay() // виртуальный пустой AlphaDecay ()
   {
      SetAtomicNumber(GetAtomicNumber() - 2);
      SetIsotopeNumber(GetIsotopeNumber() - 4);
      UpdateHalfLife();
   }

   // Атом испускает электрон.
   // Нейтрон превращается в протон.
   virtual void BetaDecay()  // виртуальный пустой BetaDecay()
```

```
    {
        SetAtomicNumber(GetAtomicNumber() + 1);
        UpdateHalfLife();
    }

    virtual double GetHalfLife()
    // виртуальный двойной точности GetHalfLife()
    {
        return half_life;     // возвратить half_life;
    }
};
```

В схеме программы в конечном счете нужно иметь цикл, представляющий увеличение времени, и на каждом шаге нужно "бросать кости", чтобы увидеть, распадается ли каждый из атомов. Если атом распадается, то нужно вызвать соответствующий метод распада — или бета-распад, или альфа-распад. Эти методы распада класса `RadioactiveAtom` обновят атомный номер и массовое число атома, чтобы они соответствовали новому изотопу, в который превратился распавшийся атом. В этом новом состоянии в действительности атом может все еще быть радиоактивным и может распадаться далее. И потому мы должны были обновить период полураспада на этом этапе выполнения программы. В следующих разделах мы продолжим разрабатывать этот пример.

В предыдущих разделах демонстрировались объявления и использование составных управляемых типов, включая ссылочные классы, классы значений, управляемые массивы, классы перечислений и классы интерфейса. В следующем разделе вы изучите функции, которые моделируют отношение "имеет" между объектами: свойства, делегирование (передача функций) и события.

Элементы, моделирующие отношение "имеет"

Вероятно, к настоящему времени вы обратили внимание, что в нашем классе `Atom` (Атом) есть много методов, которые начинаются с `Get` (Получить) и `Set` (Установить), которые позволяют моделировать связи "имеет" между объектом и свойствами объекта. Некоторые средства были добавлены в сам язык C++/CLI, чтобы зафиксировать такие часто используемые модели в языке. Внедрение этих средств в язык помогает стандартизировать часто встречающиеся фрагменты кода, это помогает создавать более читаемый код. Языковые средства в C++/CLI поддерживают связи типа "имеет", включая свойства и события.

Свойства

В языке C++/CLI предусмотрена поддержка свойств. Свойства — это элементы класса, которые представлены значением (или множеством индексированных значений в случае индексированных свойств). Многие объекты имеют свойства, и использование свойств облегчает первоклассная языковая конструкция. Несмотря на то, что сначала эта конструкция может показаться тривиальным добавлением, она действительно упрощает написание программы. Давайте заменим все методы

Get (Получить) и Set (Установить) свойствами. Для простоты мы возвратимся к примеру без интерфейсов (листинг 2.9).

Листинг 2.9. Использование свойств

```
ref class Atom              // ссылочный класс Атом
{
private:                    // приватный:
   array<double>^ pos;      // массив <двойная точность>^ pos;

public: // общедоступный:

   Atom(double x, double y, double z, unsigned int a, unsigned int n)
      // Атом (двойная точность x, двойная точность y,
      // двойная точность z, int без знака a, int без знака n),
   {
      pos = gcnew array<double>(3);
      // pos = gcnew массив <двойная точность> (3);
      pos[0] = x; pos[1] = y; pos[2] = z;

      AtomicNumber = a;
      IsotopeNumber = n;
   }

   property unsigned int AtomicNumber;
   // свойство int без знака AtomicNumber;
   property unsigned int IsotopeNumber;
   // свойство int без знака IsotopeNumber;

   property Element ElementType // свойство Элемент ElementType
   {
      Element get()             // Элемент получить()
      {
         return safe_cast<Element>(AtomicNumber);
         // возвратить safe_cast<Элемент> (AtomicNumber);
      }
      void set(Element element)
      // пусто установить(Элемент элемент)
      {
         AtomicNumber = safe_cast<int>(element);
         // AtomicNumber = safe_cast<int>(элемент);
      }
   }

   property double Position[int]
   // свойство двойная точность Позиция[int]
   {
      // Если индекс окажется вне диапазона, то при доступе к массиву
      // будет вызвано исключение IndexOutOfRange.
      double get(int index)          {
      // двойная точность получить(int индекс) {
         return pos[index]; // возвратить pos[индекс];
      }
```

```
        void set(int index, double value)
        // пусто установить(int индекс, двойная точность значение)
        {
            pos[index] = value; // pos[индекс] = значение;
        }
    }
};
```

Мы создаем четыре свойства: `AtomicNumber`, `IsotopeNumber`, `ElementType` и `Position` (Позиция). Мы преднамеренно используем три различных способа определения этих свойств, чтобы пояснить, что можно делать со свойствами. Свойство `ElementType` определено стандартно, с помощью обычно используемой конструкции. Сначала идет имя (название) свойства, за его именем следует блок, содержащий методы `get` (получить) и `set` (установить), полностью прототипированные и реализованные. Имена средства доступа (аксессоры) должны быть `get` (получить) и `set` (установить), хотя реализовывать их оба не обязательно. Если реализовать только один из этих методов, свойство становится доступным только для чтения или только для записи. Свойства `AtomicNumber` и `IsotopeNumber` — это то, что известно как тривиальные свойства. Тривиальные свойства имеют методы (механизмы) для чтения и установки, создаваемые для них автоматически; также обратите внимание на то, что мы удаляем поля `atomicNumber` и `isotopeNumber`. Они больше не нужны, так как для тривиальных свойств приватные поля создаются автоматически. Третий тип свойства известен как *индексированное* или *векторное свойство*. Неиндексированные свойства называются *скалярными свойствами*. Индексированное свойство `Position` (Позиция) реализуется тем, что похоже на массив, поскольку используются синтаксические конструкции с индексом. Векторные свойства принимают значение в квадратных скобках и используют его как индекс, чтобы определить, какое значение возвратить.

Также обратите внимание, что мы используем имена свойства точно так же, как поля в остальной части тела класса. Именно это делает свойства настолько удобными. В выражениях присваивания методы `get` (получить) и `set` (установить) свойства вызываются неявно, когда считывается значение свойства или свойству присваивается значение.

```
AtomicNumber = safe_cast<int>(element);
// AtomicNumber = safe_cast<int>(элемент);
// set (установить) вызывается неявно
```

Вы можете также использовать составной оператор присваивания (`+=`, `-=` и т.д.) и постфиксные или префиксные операторы для свойств, если хотите упростить синтаксис. Рассмотрим, например, метод `AlphaDecay` в классе `RadioactiveAtom`. Его можно написать так, как показано в листинге 2.10.

Листинг 2.10. Использование составных операторов присваивания для свойств

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

Делегаты и события

Управляемые типы могут иметь дополнительные конструкции для событий. События основаны на делегировании (передаче функций), т.е. на управляемых типах, которые походят на крутые указатели функций. Делегаты на самом деле не просто указатели функций, потому что они могут ссылаться на целое множество методов, а не только на один метод. Кроме того, ссылаясь на нестатический метод, они ссылаются и на объект, и на метод, чтобы обратиться к нужному объекту. Как вы увидите позже, синтаксические конструкции и для определения делегата, и его вызова, более просты, чем соответствующие синтаксические конструкции для использования указателя функции или указателя на член. Продолжая решать задачу о радиоактивности, мы будем теперь использовать делегирование (передачу функций) для реализации радиоактивного распада. Атомы имеют определенную вероятность распада. Вероятность распада в течение определенного времени может быть определена по периоду полураспада с помощью формулы

вероятность распада = ln 2 / *период полураспада* * *timestep*.

Константа λ, известная как постоянная распада, представляет собой величину ln 2 / *период полураспада*; таким образом, эту формулу можно также записать и так:

вероятность распада = λ * *timestep*.

Листинг 2.11 демонстрирует создание типа делегата, в данном случае `DecayProcessFunc`. Класс `RadioactiveAtoms` имеет свойство по имени `DecayProcess`, которое имеет тип данного делегата. Это свойство может быть установлено (`set`) равным функции бета-распада (здесь бета-минус-распад), функции альфа-распада или, возможно, некоторому другому редкому типу радиоактивного распада.

Делегат указывает и объект, и метод. В этом состоит основное различие между делегированием (передачей функций) и указателем на функцию-член в классическом C++. Листинг 2.11 содержит полный код, причем код делегата выделен подчеркиванием. Я удалил интерфейс, который использовался в листинге 2.8, поскольку он теперь не является центральным в обсуждении.

Листинг 2.11. Использование делегатов

```
// radioactive_decay.cpp
using namespace System;
// используем пространство имен Система;

// Это объявление типа делегата, который не имеет параметров.
delegate void DecayProcessFunc();
// делегат пусто DecayProcessFunc();

enum class Element;          // enum класс Элемент;
// тот же самый, что и прежде
ref class Atom;              // ссылочный класс Атом;
// тот же самый, что и прежде, но без текущих координат


ref class RadioactiveAtom : Atom
// ссылочный класс RadioactiveAtom: Атом
```

```
{

public:                    // общедоступный:
    RadioactiveAtom(int a, int n, bool is_stable, double half_life)
    // RadioactiveAtom (int a, int n, bool is_stable,
    //    двойная точность half_life)
        : Atom(a, n)         // : Атом (a, n)
    {
        IsStable = is_stable;
        HalfLife = half_life; // Период полураспада = half_life;
        Lambda = Math::Log(2) / half_life;
        // Лямбда = Математика::Log(2) / half_life;
    }

    // Атом выбрасывает альфа-частицу
    // и потому он теряет два протона и два нейтрона.
    virtual void AlphaDecay() // виртуальный пустой AlphaDecay()
    {
        AtomicNumber -= 2;
        IsotopeNumber -= 4;
        Update();              // Обновить();
    }

    // Атом испускает электрон.
    void BetaDecay()          // пусто BetaDecay ()
    {
        AtomicNumber++;
        Update();             // Обновить();
    }

    property bool IsStable;   // свойство bool IsStable;
    property double HalfLife;
    // свойство двойная точность Период полураспада;
    property double Lambda;   // свойство двойная точность Лямбда;
    void Update()             // пусто Обновить()
    {
        // В данном случае мы предполагаем, что в результате
        // распада образуется устойчивое ядро.
        // nullptr — способ в C++/CLI указать отсутствие значения
        // у дескриптора.
        DecayProcess = nullptr;
        IsStable = true;      // IsStable = истина;
    }

    // Объявить свойство делегата.
    // Мы вызываем его, когда атом распадается.
    property DecayProcessFunc^ DecayProcess;
    // свойство DecayProcessFunc^ DecayProcess;

}; // ссылочный класс (ref class) RadioactiveAtom
```

```
void SimulateDecay(int a, int n, double halflife, int step,
// пусто SimulateDecay (int a, int n,
// двойная точность период полураспада, int шаг,
                int max_time, int num_atoms, int seed)
        // int max_time, int num_atoms, int начальное число)
{
    array<RadioactiveAtom^>^ atoms =
              gcnew array<RadioactiveAtom^>(num_atoms);
    // массив<RadioactiveAtom^>^ атомы =
    // gcnew массив<RadioactiveAtom^>(num_atoms);

    // Инициализировать массив.
    // Мы здесь не можем использовать оператор for each
    // (для каждого), потому что оператор for each
    // (для каждого) не может изменить массив атомов.
    for (int i = 0; i < num_atoms; i++) // обычный оператор для
    {
        atoms[i] = gcnew RadioactiveAtom(a, n, false, halflife);
        // атомы[i] = gcnew RadioactiveAtom(a, n, ложь,
        // период полураспада);
        // Создать делегата.
        atoms[i]->DecayProcess = // атомы [i]-> DecayProcess =
          gcnew DecayProcessFunc(atoms[i], &RadioactiveAtom::BetaDecay);
     // gcnew DecayProcessFunc (атомы[i], &RadioactiveAtom::BetaDecay);
    }

    Random^ rand = gcnew Random(seed);
    // Случайный^ rand = gcnew Случайный(начальное число);
    for (int t = 0; t < max_time; t++)
    // для (int t = 0; t <max_time; t ++)
    {
        for each (RadioactiveAtom^ atom in atoms)
        // для каждого (RadioactiveAtom^ атом в атомах)
        {
            if ((!atom->IsStable) && atom->Lambda * step > rand->NextDouble())
// если ((! атом->IsStable), && атом->Лямбда * шаг > rand->NextDouble())
            {
                // Вызвать делегата.
                atom->DecayProcess->Invoke();
                // атом->DecayProcess->Вызвать();
            }
        }
    }
}


int main()                  // int главная программа()
{
    // Углерод 14. Атомный номер 6. Массовое число 14
    // Период полураспада 5730 лет
    // Количество атомов 10000
```

```
    // Максимальное время 10000
    // Начальное значение случайного числа 7757
    SimulateDecay(6, 14, 5730, 1, 10000, 10000, 7757);
}
```

Код делегата состоит из объявления делегата, в котором указываются параметры и возвращаемый тип делегата. Далее, существует момент, когда делегат создается. Делегат — ссылочный тип; таким образом, обращаться к нему нужно с помощью дескриптора, причем для создания делегата следует использовать `gcnew`. Если делегат должен сослаться на нестатическую функцию-член, нужно вызвать конструктор делегата, который принимает и указатель объекта, и метод, который вызывается; при этом используется операция вычисления адреса (`&`) и квалифицированное имя метода. Если делегат назначается статическому методу, объект опускается, а передается только второй параметр, указывающий метод, вот так:

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

Пока мы использовали делегирование, но не события. Событие — это абстракция, предназначенная для представления того, что кое-что происходит. Методы, называемые обработчиками события, могут быть присоединены к событиям, чтобы некоторым способом ответить на событие. События имеют тип делегата, но как вы уже видели, сами делегаты могут использоваться независимо от событий. Делегат формирует связь между источником события (возможно, это действие пользователя или некоторое действие, инициализированное другим кодом) и объектом и функцией, которая в некотором роде отвечает на действие. В нашем случае класс `RadioactiveAtom` будет иметь событие `Decay` (Распад), объявленное в листинге 2.12.

Листинг 2.12. Объявление события

```
ref class RadioactiveAtom    // ссылочный класс RadioactiveAtom
{
    // другой код...

    // объявление события
    event DecayProcessFunc^ Decay;
    // событие DecayProcessFunc^ Распад;

};
```

Вместо того чтобы непосредственно вызвать делегата, мы вызываем событие в клиентском коде, используя синтаксические конструкции для вызова функции. Код, который соединяет событие, выглядит так же, как свойство делегата (листинг 2.13).

Листинг 2.13. Присоединение и инициирование события

```
// Присоединить событие.
atoms[i]->Decay += // атомы[i]-> Распад +=
gcnew DecayProcessFunc(atoms[i], &RadioactiveAtom::BetaDecay);
// gcnew DecayProcessFunc (атомы[i], &RadioactiveAtom::BetaDecay);
```

```
// ...

// Запустить событие.
a->Decay(); // Распад;
```

Событие может вызвать множество действий. Вы узнаете о таких возможностях в главе 7 "Средства классов .NET".

Вы можете, конечно, усовершенствовать этот проект. Возможно, вы обеспокоены тем фактом, что каждый экземпляр класса `RadioactiveAtom` содержит его собственный период полураспада и свойство `lambda`. Вместо этого можно создать статическую структуру данных, чтобы хранить информацию о периоде полураспада для каждого типа изотопа. На что была бы похожа эта структура? Для поиска в ней требовалось бы два индекса — атомный номер и изотоп. Однако решение в виде двумерного массива привело бы к огромному бесполезному перерасходу памяти, так как большинство клеток (ячеек) никогда не будет использоваться. Но можно пробовать реализовать таблицу изотопов в виде разреженного массива — структуры данных, которая может использоваться как массив, но на самом деле представляет собой хэш-таблицу, чтобы избежать перерасхода памяти для хранения неиспользуемых элементов. Реализация такого типа коллекции, вероятно, была бы шаблонной в классическом C++. В C++/CLI это может быть шаблон или это другой вид параметризованного типа — родовой тип, который описан в следующем разделе.

Родовые типы

В то время как шаблоны поддерживаются в C++/CLI и для родных, и для управляемых типов, в C++/CLI был определен еще один вид параметризованного типа — родовые (общие) объекты. Родовые объекты имеют различные предназначения, поддерживая параметризацию типов во время выполнения, тогда как шаблоны поддерживают параметризацию типов во время компиляции. Вы узнаете, во что выливается это различие, в главе 11 "Параметризованные функции и типы". .NET Framework 2.0 поддерживает родовые объекты и предоставляет родовые классы коллекций. Один из таких классов — родовой список `List`, который является классом динамических массивов, которые расширяются автоматически, чтобы в них можно было разместить большие количества элементов. Определение класса `List` (Список) выглядит примерно так, как в коде в листинге 2.14.

Листинг 2.14. Определение родового класса **`List`**

```
generic <typename T>          // родовой <typename T>
ref class List
{
public:                     // общедоступный:
   T Add(T t) { /*...*/ }
   void Remove(T t) { /*...*/ }
   // другие методы
};
```

Это объявление указывает, что `List` (Список) — родовой (общий) тип с одним параметром типа, `T`. Возвращаясь к нашему примеру с изотопами химических элементов, класс `List` представляет собой хороший выбор для представления изотопов элемента, так как каждый элемент имеет различное количество изотопов. Родовая коллекция `List` представляет собой свойство в этом классе. В объявлении объекта `List` фактический тип (дескриптор для `Isotope` (Изотоп)) используется как параметр. Дескрипторы, а не сами ссылочные типы используются при подстановке параметра типа. Можно также использовать тип значений и без дескриптора. В листинге 2.15 приведен класс `ElementType` со свойством `Isotopes` (Изотопы), которое является списком изотопов конкретного элемента.

Листинг 2.15. Ссылочный класс, в котором используется родовой список **`List`** как свойство

```
ref class Isotope;          // реализация опущена ради краткости
                            // ссылочный класс Изотоп;

ref class ElementType       // ссылочный класс ElementType
{

    // другие свойства, определяющие имя элемента, атомный номер и т.д.

    property List<Isotope^>^ Isotopes;
    // свойство список<Изотоп^>^ Изотопы;
};
```

Использовать этот родовой тип столь же просто, как и тип управляемого массива. Код в листинге 2.16 использует оператор `for each` (для каждого), чтобы при проходе через родовую коллекцию `List` выполнить поиск изотопа по его номеру. Предположите, что класс `Isotope` имеет свойство `IsotopeNumber`.

Листинг 2.16. Просмотр родовой коллекции

```
ref class ElementType        // ссылочный класс ElementType
{
    // опускаем другие члены класса

    // Найти изотоп по номеру. Если не найден, возвратить
    // нулевой (пустой) дескриптор (nullptr).
    Isotope^ FindIsotope(int isotope_number)
    // Изотоп^ FindIsotope(int isotope_number)
    {
        for each (Isotope^ isotope in Isotopes)
        // для каждого (Изотоп^ изотоп в Изотопах)
        {
            if (isotope->IsotopeNumber == isotope_number)
            // если (изотоп->IsotopeNumber == isotope_number)
                return isotope; // возвратить изотоп;
        }
        return nullptr;       // возвратить nullptr;
    }
};
```

Более полное обсуждение родовых объектов и управляемых шаблонов — тема главы 11 “Параметризованные функции и типы”. Ну а мы будем использовать родовой класс `List` в примерах этой книги повсюду. Позже в главе 11 “Параметризованные функции и типы” вы научитесь определять родовые классы и функции.

Резюме

В этой главе вы изучили некоторые важные конструкции языка C++/CLI. Конечно, есть и другие важные средства, и о каждом из них можно рассказать намного больше. Но вам был представлен краткий обзор примитивных типов, различных составных типов, управляющих массивов, свойств, делегатов, событий и параметризованных типов. В последующих главах мы рассмотрим каждый из этих аспектов языка более подробно.

Но перед этим научимся компилировать и строить программы на C++/CLI.