6

C#, Visual Basic и C++/CLI

C# — это язык программирования, спроектированный специально для .NET. Существует более 50 языков для написания приложений .NET, например, Eiffel, Smalltalk, COBOL, Haskell, Pizza, Pascal, Delphi, Oberon, Prolog, Ruby и многие другие. Лишь только одна корпорация Microsoft поставляет языки C#, Visual Basic, C++/CLI, J# и JScript.NET.

Каждый язык обладает своими достоинствами и недостатками; некоторые вещи легко делать на одном языке, но сложно — на другом. Классы .NET Framework всегда одни и те же, но синтаксис языка абстрагирует различные средства .NET Framework. Например, оператор `using` в C# облегчает использование объектов, реализующих интерфейс `IDisposable`. На других языках может понадобиться больше кода для реализации той же функциональности.

Наиболее часто используемыми языками .NET от Microsoft являются C# и Visual Basic. C# был изначально разработан для .NET, почерпнув идеи из C++, Java, Pascal и других языков. Visual Basic уходит корнями в Visual Basic 6, и был расширен объектно-ориентированными средствами для .NET.

C++/CLI — это расширение C++, соответствующее стандарту ECMA (ECMA 372). Значительное преимущество C++/CLI состоит в способности смешивать "родной" код с управляемым кодом. Вы можете расширить существующие приложения, которые используют "родной" C++, добавив к ним функциональность .NET, а также вы можете добавлять классы .NET к "родным" библиотекам, чтобы их можно было использовать из других языков .NET, таких как C#. Также возможно писать полностью управляемые приложения на C++/CLI.

Эта глава посвящена переводу приложений .NET с одного языка на другой. Если вы увидите пример кода на Visual Basic или C++/CLI, то легко сможете отобразить их на C#, и наоборот.

Ниже перечислены темы, которые рассматриваются в этой главе:

- пространства имен;
- определение типов;
- методы;
- массивы;
- управляющие операторы;

- циклы;
- обработка исключений;
- наследование;
- управление ресурсами;
- делегаты;
- события;
- обобщения;
- запросы LINQ;
- смешивание "родного" и управляемого кода в C++/CLI.

Здесь предполагается наличие у вас знаний языка C#, а также ознакомление с несколькими первыми главами настоящей книги. Знать Visual Basic и C++/CLI не обязательно.

Пространства имен

Типы .NET организованы в пространства имен. Синтаксис определения и использования пространств имен в этих трех языках существенно отличается.

Чтобы импортировать пространства имен, C# использует ключевое слово `using`. C++/CLI полностью опирается на синтаксис C++ с оператором `using namespace`. В Visual Basic для импорта пространств имен предусмотрено ключевое слово `Imports`.

В C# вы можете определять псевдонимы для классов. В C++/CLI и Visual Basic псевдонимы могут ссылаться только на другие пространства имен, но не на классы. C++ требует ключевого слова `namespace` для определения псевдонима, и то же ключевое слово используется для определения пространства имен. В Visual Basic, опять-таки, применяется ключевое слово `Imports`.

Для определения пространств имен все три языка используют ключевое слово `namespace`, но и здесь имеются отличия. В C++/CLI вы не можете определять иерархические пространства имен в одном операторе `namespace`; вместо этого пространства имен должны быть вложены друг в друга. Есть одно существенное отличие в настройках проекта: объявление пространства имен в настройках проекта C# определяет пространство имен по умолчанию, которое появляется в коде всех новых элементов, добавляемых в проект. В настройках проекта Visual Basic вы определяете корневое пространство имен, используемое всеми элементами проекта. Пространства имен, объявленные в исходном коде, определяют только подпространства внутри этого корневого пространства имен.

```
// C#
using System;
using System.Collections.Generic;
using Assm = Wrox.ProCSharp.Assemblies;
namespace Wrox.ProCSharp.Languages
{
}
// C++/CLI
using namespace System;
using namespace System::Collections::Generic;
namespace Assm = Wrox.ProCSharp.Assemblies;
namespace Wrox
{
  namespace ProCSharp
  {
```

```
        namespace Languages
        {
        }
    }
}
' Visual Basic
Imports System
Imports System.Collections.Generic
Imports Assm = Wrox.ProCSharp.Assemblies
Namespace Wrox.ProCSharp.Languages
End Namespace
```

Определение типов

.NET различает типы ссылочные и типы значений. В C# ссылочные типы определяются классами, а типы значений — структурами. Помимо того, как создавать ссылочные и типы значений, в этой главе мы также покажем, как определяются интерфейсы (ссылочные типы) и перечисления (типы значений).

Ссылочные типы

Для объявления ссылочного типа в C# и Visual Basic предусмотрено ключевое слово `class`. В C++/CLI класс и структура — это почти одно и то же; вы не должны отличать ссылочный тип от типа значений, как это делается в C# и Visual Basic. C++/CLI имеет ключевое слово `ref` для определения управляемого класса. Вы можете создавать ссылочный тип, определяя `ref class` или `ref struct`.

И в C#, и в C++/CLI класс заключается в фигурные скобки. В C++/CLI не забывайте ставить точку с запятой в конце объявления класса. В Visual Basic в конце класса помещается оператор `End Class`.

```
// C#
public class MyClass
{
}
// C++/CLI
public ref class MyClass
{
};
' Visual Basic
Public Class MyClass
End Class
```

При использовании ссылочного типа переменная должна быть объявлена, а объект должен быть распределен в управляемой куче. При объявлении дескриптора ссылочного типа C++/CLI определяет операцию ^, которая отчасти похожа на указатель * в C++. Операция `gcnew` распределяет память в управляемой куче. Используя C++/CLI, можно также объявить переменную локально, но для ссылочных типов объект все равно сохраняется в управляемой куче. В Visual Basic объявление переменной начинается с оператора `Dim`, за которым следует имя переменной. С помощью операции `new` и указания объектного типа переменная распределяется в управляемой куче.

```
// C#
MyClass obj = new MyClass();
// C++/CLI
MyClass^ obj = gcnew MyClass();
MyClass obj2;
' Visual Basic
Dim obj as New MyClass()
```

Если ссылочный тип не ссылается на память, все три языка используют разные ключевые слова: C# определяет литерал `null`, C++/CLI — `nullptr` (`NULL` допускается только с "родными" объектами), а Visual Basic — `Nothing`.

Предопределенные ссылочные типы перечислены в табл. Б.1. В C++/CLI не определены строковые и объектные типы, как это делается в других языках. Разумеется, вы можете использовать классы, определенные в .NET Framework.

Таблица Б.1. Предопределенные ссылочные типы

Тип .NET	C#	C++/CLI	Visual Basic
`System.Object`	`object`	Не определено	`Object`
`System.String`	`string`	Не определено	`String`

Типы значений

Для объявления типа значения в C# используется ключевое слово `struct`, в C++/CLI — ключевое слово `value`, а в Visual Basic — `Structure`.

```
// C#
public struct MyStruct
{
}
// C++/CLI
public value class MyStruct
{
};
' Visual Basic
Public Structure MyStruct
End Structure
```

В C++/CLI вы можете распределить тип значения в стеке, в "родной" куче с использованием операции `new` и в управляемой куче с помощью операции `gcnew`. C# и Visual Basic не предоставляют таких опций, но эти опции важны, когда "родной" и управляемый код смешиваются в C++/CLI.

```
// C#
MyStruct ms;
// C++/CLI
MyStruct ms1;
MyStruct* pms2 = new MyStruct();
MyStruct^ hms3 = gcnew MyStruct();
' Visual Basic
Dim ms as MyStruct
```

Предопределенные типы значений для разных языков перечислены в табл. Б.2. В C++/CLI тип `char` имеет размер в 1 байт для символа ASCII. В C# `char` имеет размер 2 байта для символов Unicode; то же самое в C++/CLI называется `wchar_t`. Стандарт ANSI для C++ определяет только `short <= int <= long`. На 32-разрядных машинах и `int`, и `long` имеют размер 32 бита. Чтобы определить 64-битную переменную в C++, нужно объявить ее как `long long`.

Выведение типа

C# 3.0 позволяет определять локальную переменную без явного объявления типа — с помощью ключевого слова `var`. Тип выводится из присваиваемого начального значения. Visual Basic предоставляет то же средство, используя ключевое слово `Dim`, если включена опция *Option infer*.

Таблица Б.2. Предопределенные типы значений

Тип .NET	C#	C++/CLI	Visual Basic	Размер
Char	char	wchar_t	Char	2 байта
Boolean	bool	Bool	Boolean	1 байт, содержит true или false
Int16	short	short	Short	2 байта
UInt16	ushort	unsigned short	UShort	2 байта без знака
Int32	int	int	Integer	4 байта
UInt32	uint	unsigned int	UInteger	4 байта без знака
Int64	long	long long	Long	8 байт
UInt64	ulong	unsigned long	ULong	8 байт без знака

Это может быть сделано с помощью ключа компилятора /optioninfer+ или на странице конфигурации проекта в Visual Studio.

```
// C#
var x = 3;
' Visual Basic
Dim x = 3
```

Интерфейсы

Определение интерфейсов очень похоже во всех трех языках. Все они используют ключевое слово interface:

```
// C#
public interface IDisplay
{
  void Display();
}
// C++/CLI
public interface class IDisplay
{
  void Display();
};
' Visual Basic
Public Interface IDisplay
Sub Display
End Interface
```

Реализация интерфейсов отличается. В C# и C++/CLI используется двоеточие после имени класса, за которым следует имя интерфейса. Методы, определенные в интерфейсе, реализуются. В C++/CLI методы должны быть объявлены виртуальными (virtual). Для реализации интерфейса в Visual Basic используется ключевое слово Implements, и методы, определенные интерфейсом, также должны сопровождаться ключевым словом Implements.

```
// C#
public class Person : IDisplay
{
   public void Display()
   {
   }
}
```

```
// C#, явная реализация интерфейса
public class Person : IDisplay
{
   void IDisplaey.Display()
   {
   }
}
// C++/CLI
public ref class Person : IDisplay
{
public:
   virtual void Display();
};
' Visual Basic
Public Class Person
Implements IDisplay
Public Sub Display Implements IDisplay.Display
End Sub
End Class
```

Перечисления

Перечисления определяются сходным образом во всех трех языках с помощью ключевого слова `enum` (только в Visual Basic для разделения элементов используется перевод строки, а не запятая).

```
// C#
public enum Color
{
   Red, Green, Blue
}
// C++/CLI
public enum class Color
{
   Red, Green, Blue
};
' Visual Basic
Public Enum Color
Red
Green
Blue
End Enum
```

Методы

Методы всегда объявляются внутри класса. Синтаксис C++/CLI очень похож на C# за исключением того, что модификатор доступа не является частью объявления метода, а записывается перед ним. Модификатор доступа должен завершаться двоеточием. В Visual Basic для определения метода используется ключевое слово `Sub`.

```
// C#
public class MyClass
{
  public void Foo()
  {
  }
}
// C++/CLI
public ref class MyClass
{
```

```
public:
  void Foo()
  {
  }
};
' Visual Basic
Public Class MyClass
Public Sub Foo
End Sub
End Class
```

Параметры методов и типы возврата

В C# и C++/CLI параметры, передаваемые методам, определены внутри скобок. Тип параметра объявляется перед именем переменной. Для возврата значения метод определяется с типом возврата вместо слова `void`.

В Visual Basic используются операторы `Sub` для объявления метода, не возвращающего значение, и `Function` — для метода, который должен вернуть значение. Тип возврата следует после имени метода и скобок. Visual Basic также имеет другой порядок объявления переменной и типа в параметре. Тип следует за переменной, что противоположно порядку, принятому в C# и C++/CLI.

```
// C#
public class MyClass
{
  public int Foo(int i)
  {
     return 2 * i;
  }
}
// C++/CLI
public ref class MyClass
{
public:
  int Foo(int i)
  {
     return 2 * i;
  }
};
' Visual Basic
Public Class MyClass
Public Sub Foo1(ByVal i as Integer)
End Sub
Public Function Foo(ByVal i As Integer) As Integer
  Return 2 * i
End Sub
End Class
```

Модификаторы параметров

По умолчанию типы значений передаются по значению, а ссылочные типы — по ссылке. Если тип значений, переданный в виде параметра, должен быть изменен внутри вызванного метода, в C# вы можете использовать модификатор параметра `ref`.

В C++/CLI определена операция управляемой ссылки `%`. Эта операция подобна операции ссылки `&` в C++, но с тем отличием, что `%` может использоваться с управляемыми типами, и сборщик мусора может отслеживать такие объекты в случае, если они будут перемещаться внутри управляемой кучи.

В Visual Basic для передачи параметров по ссылке применяется ключевое слово `ByRef`.

```
// C#
public class ParameterPassing
{
  public void ChangeVal(ref int i)
  {
     i = 3;
  }
}
// C++/CLI
public ref class ParameterPassing
{
public:
  int ChangeVal(int% i)
  {
     i = 3;
  }
};
' Visual Basic
Public Class ParameterPassing
Public Sub ChangeVal(ByRef i as Integer)
  i = 3
End Sub
End Class
```

При вызове метода со ссылочным параметром, только язык C# требует применения модификатора параметра. C++/CLI и Visual Basic не делают разницы при вызове метода с модификатором параметра или без него. C# здесь имеет преимущество, сразу заметное при вызове метода, который может менять значение переданного ему параметра.

Из-за синтаксиса вызова, который ничем не отличается, в Visual Basic не допускается перегрузка метода простым изменением модификатора. Компилятор C++/CLI разрешает перегрузку метода с изменением модификатора, но не может компилировать вызывающий его код, поскольку выбор конкретной реализации метода при этом неоднозначен. В C# можно перегружать и использовать методы, отличающиеся только модификатором параметра, но такая практика не рекомендуется.

```
// C#
ParameterPassing obj = new ParameterPassing();
int a = 1;
obj.ChangeVal(ref a);
Console.WriteLine(a); // выводит 3
// C++/CLI
ParameterPassing obj;
int a = 1;
obj.ChangeVal(a);
Console.WriteLine(a); // выводит 3
' Visual Basic
Dim obj as new ParameterPassing()
Dim i as Integer = 1
obj.ChangeVal(i)
Console.WriteLine(i) // выводит 3
```

В C# также определено ключевое слово `out`, когда параметр просто возвращается из метода. Эта опция недоступна в C++/CLI и Visual Basic. До тех пор, пока вызывающий и вызываемый код находятся в одном домене приложений, "за кулисами" нет разницы между `out` и `ref`, и вы можете обращаться к методу, объявленному с модификатором C# `out` в Visual Basic и C++/CLI, точно так же, как с модификатором параметра `ref`. Если метод используется между разными доменами приложений, то атрибут `[out]` может быть применяться вместе с Visual Basic и C++/CLI.

Конструкторы

Как в C#, так и в C++/CLI конструктор имеет имя, совпадающее с именем класса. Visual Basic использует процедуру по имени `New`. Ключевые слова `this` и `Me` используются для доступа к членам данного экземпляра. При вызове одного конструктора из другого в C# требуется инициализация членов. В C++/CLI и Visual Basic можно вызывать конструктор как метод.

```
// C#
public class Person
{
public Person()
  : this("unknown", "unknown")
  { }
  public Person(string firstname, string lastname)
  {
       this.firstname = firstname;
       this.lastname = lastname;
  }
  private string firstname;
  private string lastname;
}
// C++/CLI
public ref class Person
{
public:
  Person()
  {
       Person("unknown", "unknown");
  }
  Person(String^ firstname, String^ lastname)
  {
    this->firstname = firstname;
    this->lastname = lastname;
  }
private:
  String^ firstname;
  String^ lastname;
};
' Visual Basic
Public Class Person
Public Sub New()
  Me.New("unknown", "unknown")
End Sub
Public Sub New(ByVal firstname As String, ByVal lastname As String)
  Me.MyFirstname = firstname
  Me.MyLastname = lastname
End Sub
Private MyFirstname As String
Private MyLastname As String
End Class
```

Свойства

Чтобы определить свойство, в C# требуется только определить средства доступа `set` и `get` внутри блока свойства. Средство доступа `set` автоматически создает значение переменной компилятором C#. В C# 3.0 также имеется новая сокращенная нотация, в которой реализация не требуется, если только простая переменная возвращается или устанавливается средствами доступа `get` и `set`. Синтаксис отличается от C++/CLI и Visual Basic. Оба эти языка имеют ключевое слово `property`, и необходимо

определять переменную value со средством доступа set. В C++/CLI для средства доступа get также требуется указание типа возврата и типа параметра для set.

C++/CLI имеет сокращенную версию записи свойства. Используя ключевое слово property, вы просто должны определить тип и имя свойства; средства доступа set и get создаются компилятором автоматически. Если ничего не требуется кроме установки и возврата значения переменной, то сокращенная версия подходит достаточно неплохо. Если же необходима более изощренная реализация средств доступа, например, нужно проверить значение или выполнить обновление, вы должны написать полный синтаксис для свойств. Проектировщики C# 3.0 позаимствовали из C++/CLI также и сокращенную нотацию.

```
// C#
public class Person
{
  private string firstname;
  public string Firstname
  {
       get { return firstname; }
       set { firstname = value; }
  }
}
// C++/CLI
public ref class Person
{
private:
  String^ firstname;
public:
  property String^ Firstname
  {
     String^ get()
     {
        return firstname;
     }
     void set(String^ value)
     {
        firstname = value;
     }
  }
  property String^ Lastname;
};
' Visual Basic
Public Class Person
Private myFirstname As String
Public Property Firstname()
  Get
     Return myFirstname
  End Get
  Set(ByVal value)
     myFirstname = value
  End Set
End Property
Private myLastname As String
Public Property Lastname()
  Get
     Return myLastname
  End Get
  Set(ByVal value)
     myLastname = value
  End Set
End Property
End Class
```

В C# и C++/CLI свойства, доступные только для чтения, имеют лишь средство доступа `get`. В Visual Basic вы должны также специфицировать модификатор `ReadOnly`. Свойства, доступные только для записи, должны быть определены с модификатором `WriteOnly` и средством `set`.

```
' Visual Basic
Public ReadOnly Property Name()
  Get
     Return myFirstname & " " & myLastname
  End Get
End Property
```

Инициализаторы объектов

В C# 3.0 и Visual Basic свойства могут инициализироваться посредством инициализатора объекта. Свойства можно инициализировать, используя фигурные скобки — подобно инициализатору массива. Синтаксис C# и Visual Basic очень похож; только Visual Basic использует ключевое слово `With`.

```
// C#
Person p = new Person() { FirstName = "Tom", LastName = "Turbo" };
' Visual Basic
Dim p As New Person With { .FirstName = "Tom", .LastName = "Turbo" }
```

Расширяющие методы

Расширяющие методы — фундамент LINQ. Как в C#, так и в Visual Basic можно создавать расширяющие методы. Однако синтаксис отличается. C# помечает расширяющие методы ключевым словом `this` у первого параметра. Visual Basic помечает расширяющий метод атрибутом `<Extension>`.

```
// C#
public static class StringExtension
{
  public static void Foo(this string s)
  {
    Console.WriteLine("Foo {0}", s);
  }
}
' Visual Basic
Public Module StringExtension
  <Extension()> _
  Public Sub Foo(ByVal s As String)
    Console.WriteLine("Foo {0}", s)
  End Sub
End Module
```

Статические члены

Статическое поле создается только один раз для всех объектов данного типа. В C# и C++/CLI для этого предусмотрено ключевое слово `static`; в Visual Basic та же функциональность представлена ключевым словом `Shared`.

Применение статических членов выполняется с использованием имени класса, за которым следует операция точки и имя статического члена. В C++/CLI для доступа к статическим членам используется операция `::`.

```
// C#
public class Singleton
{
  private static SomeData data = null;
  public static SomeData GetData()
  {
     if (data == null)
     {
        data = new SomeData();
     }
     return data;
  }
}
// использование:
SomeData d = Singleton.GetData();
// C++/CLI
public ref class Singleton
{
private:
   static SomeData^ hData;
public:
  static SomeData^ GetData()
  {
     if (hData == nullptr)
     {
        hData = gcnew SomeData();
     }
     return hData;
  }
};
// использование:
SomeData^ d = Singleton::GetData();
' Visual Basic
Public Class Singleton
Private Shared data As SomeData
Public Shared Function GetData() As SomeData
  If data is Nothing Then
     data = new SomeData()
  End If
  Return data
End Function
End Class
' использование:
Dim d as SomeData = Singleton.GetData()
```

Массивы

Массивы обсуждались в главе 5. Класс `Array` всегда стоит "за кулисами" в массивах .NET; объявление массива заставляет компилятор создать класс, унаследованный от базового класса `Array`. Когда проектировался язык C#, он унаследовал скобочный синтаксис для массивов из C++ и расширил их инициализаторами массивов.

```
// C#
int[] arr1 = new int[3] {1, 2, 3};
int[] arr2 = {1, 2, 3};
```

Если вы используете фигурные скобки в C++/CLI, то создаете "родной" массив C++, а не массив, основанный на классе `Array`. Чтобы создавать массивы .NET, в C++/CLI введено ключевое слово `array`. Это ключевое слово использует синтаксис, подоб-

ный обобщениям — с угловыми скобками. Внутри угловых скобок указывается тип элементов массива. В C++/CLI поддерживаются инициализаторы массива с тем же синтаксисом, что и в C#.

```
// C++/CLI
array<int>^ arr1 = gcnew array<int>(3) {1, 2, 3};
array<int>^ arr2 = {1, 2, 3};
```

В Visual Basic для определения массивов применяются круглые скобки. При объявлении массива в Visual Basic требуется указать номер последнего элемента вместо количества элементов. Во всех языках .NET нумерация элементов массива начинается с 0. Это справедливо и в отношении Visual Basic. Для ясности в Visual Basic 9 в объявлении массива введено выражение `0 To number`. Оно всегда начинается с 0, а `To` используется просто для лучшей читабельности.

В Visual Basic также поддерживаются инициализаторы массивов, если массив инициализируется операцией `New`.

```
' Visual Basic
Dim arr1(0 To 2) As Integer()
Dim arr2 As Integer() = New Integer(0 To 2) {1, 2, 3};
```

Управляющие операторы

Управляющие операторы указывают, какой код должен выполняться. В C# определены операторы `if` и `switch`, а также условная операция.

Оператор `if`

Оператор `if` в C# точно такой же, как и в C++/CLI. В Visual Basic вместо фигурных скобок используется `If-Then/Else/End If`.

```
// C# and C++/CLI
if (a == 3)
{
   // что-то делать
}
else
{
   // что-то делать
}
' Visual Basic
If a = 3 Then
   ' что-то делать
Else
   ' что-то делать
End If
```

Условная операция

В C# и C++/CLI поддерживается условная операция — облегченная версия оператора `if`. В C++/CLI эта операция известна под названием тернарной. Ее первый аргумент должен возвращать результат булевского типа, и если он равен `true`, то вычисляется первое выражение (второй аргумент), в противном случае — второе выражение (третий аргумент). В Visual Basic имеется функция `IIf` из библиотеки Visual Basic Runtime Library, которая поддерживает аналогичную функциональность.

```
// C#
string s = a > 3 ? "one" : "two";
```

```
// C++/CLI
String^ s = a > 3 ? "one" : "two";
' Visual Basic
Dim s As String = IIf(a > 3, "one", "two")
```

Оператор switch

Операторы switch в C# и C++/CLI выглядят очень похоже, но между ними есть существенная разница. В C# поддерживаются строки для выбора альтернатив case. Это невозможно в C++. В C++ вам придется в таком случае воспользоваться if-else. В C++/CLI поддерживается неявное сквозное прохождение от одного case к другому. В C# компилятор проверяет наличие break или goto между case. В C# неявное сквозное прохождение выполняется, только если в ветви case нет никаких операторов.

В Visual Basic имеется оператор Select Case вместо switch/case. Аналог break не только не нужен, но и невозможен. Неявное сквозное прохождение от одного case к другому невозможно, даже если нет ни одного оператора после Case; вместо этого Case может комбинироваться с помощью And, Or и To, например, 3 To 5.

```
// C#
string GetColor(Suit s)
{
  string color;
  switch (s)
  {
   case Suit.Heart:
   case Suit.Diamond:
    color = "Red";
    break;
   case Suit.Spade:
   case Suit.Club:
    color = "Black";
    break;
   default:
    color = "Unknown";
    break;
  }
  return color;
}
// C++/CLI
String^ GetColor(Suit s)
{
  String^ color;
  switch (s)
  {
   case Suit::Heart:
   case Suit::Diamond:
     color = "Red";
     break;
   case Suit::Spade:
   case Suit::Club:
     color = "Black";
     break;
   default:
     color = "Unknown";
     break;
  }
  return color;
}
```

```
' Visual Basic
Function GetColor(ByVal s As Suit) As String
Dim color As String = Nothing
Select Case s
  Case Suit.Heart And Suit.Diamond
     color = "Red"
  Case Suit.Spade And Suit.Club
     color = "Black"
  Case Else
     color = "Unknown"
End Select
Return color
End Function
```

Циклы

С помощью циклов можно многократно выполнять код до тех пор, пока удовлетворяется некоторое условие. Циклы в C# обсуждаются в главе 2, включая `for`, `while`, `do..while` и `foreach`. Все эти операторы очень похожи в C# и C++/CLI. В Visual Basic определены другие операторы.

Оператор `for`

Оператор `for` в C# и C++/CLI аналогичен. В Visual Basic вы не можете инициализировать переменную внутри оператора `For/To`, а должны это сделать предварительно. `For/To` не требует, чтобы за ним следовало слово `Step`. По умолчанию — `Step 1`. В случае если не нужно инкрементирование на 1, после `For/To` следует указать `Step`.

```
// C#
for (int i = 0; i < 100; i++)
{
   Console.WriteLine(i);
}
// C++/CLI
for (int i = 0; i < 100; i++)
{
   Console::WriteLine(i);
}
' Visual Basic
Dim count as Integer
For count = 0 To 99 Step 1
   Console.WriteLine(count)
Next
```

Операторы `while` и `do..while`

Операторы `while` и `do..while` одинаковы в C# и C++/CLI. В Visual Basic имеются очень похожие конструкции — `Do While/Loop` и `Do/Loop While`.

```
// C#
int i = 0;
while (i < 3)
{
   Console.WriteLine(i++);
}
i = 0;
do
{
   Console.WriteLine(i++);
} while (i < 3);
```

```
// C++/CLI
int i = 0;
while (i < 3)
{
   Console::WriteLine(i++);
}
i = 0;
do
{
   Console::WriteLine(i++);
} while (i < 3);
' Visual Basic
Dim num as Integer = 0
Do While (num < 3)
  Console.WriteLine(num)
  num += 1
Loop
num = 0
Do
  Console.WriteLine(num)
  num += 1
Loop While (num < 3)
```

Оператор `foreach`

Оператор `foreach` использует интерфейс `IEnumerable`. Этот оператор отсутствует в ANSI C++, но имеется в расширении ANSI C++/CLI. В отличие от `foreach` из C#, в C++/CLI пробел между `for` и `each` не ставится. Оператор `For Each` в Visual Basic не позволяет объявлять тип итератора внутри цикла; его тип должен быть объявлен заранее.

```
// C#
int[] arr = {1, 2, 3};
foreach (int i in arr)
{
   Console.WriteLine(i);
}
// C++/CLI
array<int>^ arr = {1, 2, 3};
for each (int i in arr)
{
   Console::WriteLine(i);
}
' Visual Basic
Dim arr() As Integer = New Integer() {1, 2, 3}
Dim num As Integer
For Each num In arr
   Console.WriteLine(num)
Next
```

В то время как `foreach` облегчает итерацию по коллекциям, C# поддерживает создание перечислителей посредством оператора `yield`. В Visual Basic и C++/CLI оператор `yield` не доступен. Вместо него эти языки требуют реализации интерфейсов `IEnumerable` и `IEnumerator` вручную. Оператор `yield` объясняется в главе 5.

Обработка исключений

Обработка исключений обсуждалась в главе 14. Это средство очень похоже во всех трех языках. Везде используется конструкция `try/catch/finally` для обработки исключений, а ключевое слово `throw` — для генерации исключения.

```
// C#
public void Method(Object o)
{
   if (o == null)
   throw new ArgumentException("Error");
}
public void Foo()
{
  try
  {
   Method(null);
  }
  catch (ArgumentException ex)
  { }
  catch (Exception ex)
  { }
  finally
  { }
}
// C++/CLI
public:
void Method(Object^ o)
{
  if (o == nullptr)
    throw gcnew ArgumentException("Error");
}
void Foo()
{
  try
  {
   Method(nullptr);
  }
  catch (ArgumentException^ ex)
  { }
  catch (Exception^ ex)
  { }
  finally
  { }
}
' Visual Basic
Public Sub Method(ByVal o As Object)
If o = Nothing Then
  Throw New ArgumentException("Error")
End Sub
Public Sub Foo()
Try
  Method(Nothing)
Catch ex As ArgumentException
  '
Catch ex As Exception
  '
Finally
  '
End Try
End Sub
```

Наследование

Языки, поддерживающие .NET, предоставляют множество ключевых слов, касающихся определения полиморфного поведения, переопределения или сокрытия методов и модификаторов доступа, предназначенных для разрешения или запрета доступа к членам. Для C# эта функциональность обсуждалось в главе 4, посвященной наследованию. Вся эта функциональность в C#, C++/CLI и Visual Basic очень похожа, однако ключевые слова отличаются.

Модификаторы доступа

Модификаторы доступа в C++/CLI и Visual Basic очень похожи на те, что имеются в C#, но с некоторыми заметными отличиями (табл. Б.3). В Visual Basic используется модификатор доступа `Friend` вместо `internal` для доступа к типам из одной и той же сборки. В C++/CLI имеется еще один модификатор: `protected private`. Модификатор `internal protected` позволяет получить доступ к членам класса из той же сборки, а также из других сборок, если тип наследует данный базовый тип. В C# и Visual Basic нет возможности открыть доступ только типам-наследникам из той же сборки. А в C++/CLI такое возможно, благодаря `protected private`. Ключевое слово `private` здесь говорит о том, что доступ извне данной сборки закрыт, но в пределах той же сборки `protected`-доступ возможен. Порядок слов — т.е. пишете вы `protected private` или `private protected` — не имеет значения. Модификатор доступа, позволяющий большее, всегда располагается внутри сборки, а модификатор, позволяющий меньшее — вне ее.

Таблица Б.3. Модификаторы доступа

C#	C++/CLI	Visual Basic
Public	public	Public
protected	protected	Protected
private	private	Private
internal	internal	Friend
internal protected	internal protected	Protected Friend
Не возможен	protected private	Не возможен

Ключевые слова

Ключевые слова, важные для наследования, перечислены в табл. Б.4.

Таблица Б.4. Ключевые слова, связанные с наследованием

C#	C++/CLI	Visual Basic	Функциональность
`:`	`:`	`Implements`	Реализует интерфейс.
`:`	`:`	`Inherits`	Наследует базовый класс.
`virtual`	`virtual`	`Overridable`	Объявляет метод для поддержки полиморфизма.
`overrides`	`override`	`Overrides`	Переопределяет виртуальный метод.

Окончание табл. Б.4

C#	C++/CLI	Visual Basic	Функциональность
new	new	Shadows	Скрывает метод из базового класса.
abstract	abstract	MustInherit	Абстрактный класс.
sealed	sealed	NotInheritable	Финальный класс (не наследуемый).
abstract	abstract	MustOverride	Абстрактный метод.
sealed	sealed	NotOverrideable	Финальный метод.
this	this	Me	Ссылка на текущий объект.
base	Classname::	MyBase	Ссылка на базовый класс.

Порядок размещения ключевых слов в разных языках имеет значение. В следующем примере кода объявляется абстрактный базовый класс `Base` с одним абстрактным методом и одним реализованным виртуальным методом. От класса `Base` наследуется класс `Derived`, в котором абстрактный метод реализуется, а виртуальный метод переопределяется.

```
// C#
public abstract class Base
{
  public virtual void Foo()
  {
  }
  public abstract void Bar();
}
public class Derived : Base
{
  public override void Foo()
  {
    base.Foo();
  }
  public override void Bar()
  {
  }
}
// C++/CLI
public ref class Base abstract
{
public:
  virtual void Foo()
  {
  }
  virtual void Bar() abstract;
};
public ref class Derived : public Base
{
public:
  virtual void Foo() override
  {
       Base::Foo();
  }
```

```
  virtual void Bar() override
  {
  }
};
' Visual Basic
Public MustInherit Class Base
  Public Overridable Sub Foo()
  End Sub
  Public MustOverride Sub Bar()
End Class
Public class Derived
  Inherits Base
  Public Overrides Sub Foo()
    MyBase.Foo()
  End Sub
  Public Overrides Sub Bar()
  End Sub
End Class
```

Управление ресурсами

Работа с ресурсами была описана в главе 12, как посредством интерфейса `IDisposable`, так и реализацией финализатора. В этом разделе мы покажем, как это выглядит в C++/CLI и Visual Basic.

Реализация интерфейса `IDisposable`

Для освобождения ресурсов в интерфейсе `IDisposable` определен метод `Dispose()`. Используя C# и Visual Basic, вы должны реализовать интерфейс `IDisposable`. В C++/CLI этот интерфейс также реализован, но это осуществляется компилятором, если вы написали деструктор.

```
// C#
public class Resource : IDisposable
{
  public void Dispose()
  {
    // освобождение ресурсов
  }
}
// C++/CLI
public ref class Resource
{
public:
  ~Resource()
  {
    // освобождение ресурсов
  }
};
' Visual Basic
Public Class Resource
  Implements IDisposable
  Public Sub Dispose() Implements IDisposable.Dispose
    'освобождение ресурсов
  End Sub
End Class
```

В C++/CLI метод `Dispose()` вызывается при использовании оператора `delete`.

Оператор `using`

Оператор C# `using` реализует шаблон "захватить/использовать/освободить" для освобождения ресурса, как только он более не используется — даже в случае возникновения исключения. Компилятор создает оператор `try/finally` и вызывает метод `Dispose` внутри блока `finally`. Версия Visual Basic 9 поддерживает оператор `using` подобно C#. В C++/CLI предусмотрен еще более элегантный подход к этой проблеме. Если ссылочный тип объявлен локально, то компилятор создает оператор `try/finally` для вызова метода `Dispose()` в конце блока.

```
// C#
using (Resource r = new Resource())
{
   r.Foo();
}
// C++/CLI
{
   Resource r;
   r.Foo();
}
' Visual Basic
Using r As New Resource
   r.Foo()
End Using
```

Переопределение `Finalize`

Если класс содержит "родные" ресурсы, которые должны быть освобождены, он должен переопределить метод `Finalize()` класса `Object`. В C# это делается путем написания деструктора. В C++/CLI предусмотрен специальный синтаксис с префиксом `!` для определения финализатора. Внутри финализатора не допускается освобождать содержащиеся объекты, также имеющие финализатор, поскольку четкий порядок финализации не гарантирован. Вот почему шаблон `Dispose` определяет вдобавок метод `Dispose()` с параметром булевского типа. В C++/CLI нет необходимости реализовывать этот шаблон в коде, поскольку это делается компилятором. Деструктор C++/CLI реализует методы `Dispose()`. В Visual Basic и метод `Dispose()`, и финализатор должны быть реализованы вручную. Однако большинство классов Visual Basic не используют "родные" ресурсы непосредственно, а только с помощью других классов. В Visual Basic обычно нет необходимости переопределять метод `Finalize()`, но реализация метода `Dispose()` часто необходима.

Написание деструктора на C# переопределяет метод `Finalize()` базового класса. Деструктор C++/CLI реализует интерфейс `IDisposable`.

```
// C#
public class Resource : IDisposable
{
  ~Resource // переопределение Finalize
  {
     Dispose(false);
  }
  protected virtual void Dispose(bool disposing)
  {
     if (disposing) // освободить вложенные члены
     {
     }
```

```
      // освободить ресурсы класса
      GC.SuppressFinalize(this);
    }
    public void Dispose()
    {
      Dispose(true);
    }
}
// C++/CLI
public ref class Resource
{
public:
  ~Resource() // реализует IDisposable
  {
    this->!Resource();
  }
  !Resource() // переопределение Finalize
  {
    // освободить ресурсы
  }
};
' Visual Basic
Public Class Resource
Implements IDisposable
Public Sub Dispose() Implements IDisposable.Dispose
  Dispose(True)
  GC.SuppressFinalize(Me)
End Sub
Protected Overridable Sub Dispose(ByVal disposing)
  If disposing Then
     ' освободить вложенные ресурсы
  End If
  ' освободить ресурсы данного класса
End Sub
Protected Overrides Sub Finalize()
  Try
     Dispose(False)
  Finally
     MyBase.Finalize()
  End Try
End Sub
End Class
```

Делегаты

Делегаты — безопасные в отношении типов указатели на методы — обсуждались в главе 7. Во всех трех языках для определения делегата может быть использовано ключевое слово `delegate`. Отличие заключается в использовании делегатов.

Код примера показывает класс `Demo` со статическим методом `Foo()` и методом экземпляра `Bar()`. Оба эти метода вызываются экземплярами делегатов типа `DemoDelegate`. Тип `DemoDelegate` объявлен как вызывающий метод с типом возврата `void` и одним параметром `int`.

В C# 2.0 поддерживаются предположения делегатов, когда компилятор создает экземпляр делегата и передает адрес метода.

В C# и C++/CLI два делегата могут быть комбинированы в один с помощью операции +.

```
// C#
public delegate void DemoDelegate(int x);
public class Demo
{
  public static void Foo(int x) { }
  public void Bar(int x) { }
}
Demo d = new Demo();
DemoDelegate d1 = Demo.Foo;
DemoDelegate d2 = d.Bar;
DemoDelegate d3 = d1 + d2;
d3(11);
```

Предположения делегатов невозможны в C++/CLI. В C++/CLI необходимо создать новый экземпляр типа делегата и передать адрес метода конструктору.

```
// C++/CLI
public delegate void DemoDelegate(int x);
public ref class Demo
{
public:
  static void Foo(int x) { }
  void Bar(int x) { }
};
Demo^ d = gcnew Demo();
DemoDelegate^ d1 = gcnew DemoDelegate(&Demo::Foo);
DemoDelegate^ d2 = gcnew DemoDelegate(d, &Demo::Bar);
DemoDelegate^ d3 = d1 + d2;
d3(11);
```

Подобно C++/CLI, Visual Basic не поддерживает предположения делегатов. Вы должны создать новый экземпляр типа делегата и передать адрес методу. Visual Basic имеет оператор `AddressOf` для передачи адреса методу.

Visual Basic не перегружает операцию + для делегатов, поэтому вместо этого необходимо вызывать метод `Combine()` из класса `Delegate`. Класс `Delegate` пишется внутри квадратных скобок, поскольку `Delegate` — ключевое слово в Visual Basic, а потому невозможно использовать класс с таким же именем. Помещение `Delegate` в квадратные скобки гарантирует, что будет использован класс `Delegate` вместо ключевого слова `Delegate`.

```
' Visual Basic
Public Delegate Sub DemoDelegate(ByVal x As Integer)
Public Class Demo
  Public Shared Sub Foo(ByVal x As Integer)
    '
  End Sub
  Public Sub Bar(ByVal x As Integer)
    '
  End Sub
End Class
Dim d As New Demo()
Dim d1 As New DemoDelegate(AddressOf Demo.Foo)
Dim d2 As New DemoDelegate(AddressOf d.Bar)
Dim d3 As DemoDelegate = [Delegate].Combine(d1, d2)
d3(11)
```

События

С ключевым словом `event` может быть реализован механизм подписки на основе делегатов. Во всех языках предусмотрено ключевое слово `event` для предоставления

событий из класса. Класс `EventDemo` возбуждает событие по имени `DemoEvent` типа `DemoDelegate`.

В C# синтаксис для инициации события выглядит как вызов метола события. Переменная события остается равной `null` до тех пор, пока никто не зарегистрирует событие, поэтому следует выполнять проверку на неравенство `null`, прежде чем возбуждать событие. Метод-обработчик регистрируется операцией +=, и адрес этого метода передается с помощью механизма предположения делегата.

```
// C#
public class EventDemo
{
  public event DemoDelegate DemoEvent;
  public void FireEvent()
  {
     if (DemoEvent != null)
       DemoEvent(44);
  }
}
public class Subscriber
{
  public void Handler(int x)
  {
     // handler implementation
  }
}
//...
EventDemo evd = new EventDemo();
Subscriber subscr = new Subscriber();
evd.DemoEvent += subscr.Handler;
evd.FireEvent();
```

C++/CLI очень похож на C#, за исключением того, что возбуждение события не требует предварительной проверки переменной события на неравенство `null`. Это делается автоматически в коде IL, созданном компилятором.

И в C#, и в C++/CLI используется операция -= для отмены регистрации события.

```
// C++/CLI
public ref class EventDemo
{
public:
  event DemoDelegate^ DemoEvent;
  public void FireEvent()
  {
     DemoEvent(44);
  }
}
public class Subscriber
{
public:
  void Handler(int x)
  {
     // реализация обработчика
  }
}
//...
EventDemo^ evd = gcnew EventDemo();
Subscriber^ subscr = gcnew Subscriber();
evd->DemoEvent += gcnew DemoDelegate(subscr, &Subscriber::Handler);
evd->FireEvent();
```

Visual Basic поддерживает другой синтаксис. Событие объявляется ключевым словом `Event` — точно так же, как и в C# и C++/CLI. Однако событие возбуждается оператором `RaiseEvent`. Оператор `RaiseEvent` проверяет, инициализирована ли подписчиком переменная события. Для регистрации обработчика служит оператор `AddHandler`, имеющий ту же функциональность, что и операция += в C#. `AddHandler` требует двух параметров: первый определяет событие, а второй — адрес обработчика. Оператор `RemoveHandler` используется для отмены регистрации обработчика события.

```
' Visual Basic
Public Class EventDemo
  Public Event DemoEvent As DemoDelegate
  public Sub FireEvent()
     RaiseEvent DemoEvent(44);
  End Sub
End Class
Public Class Subscriber
  Public Sub Handler(ByVal x As Integer)
     ' handler implementation
  End Sub
End Class
'...
Dim evd As New EventDemo()
Dim subscr As New Subscriber()
AddHandler evd.DemoEvent, AddressOf subscr.Handler
evd.FireEvent()
```

Visual Basic предлагает другой синтаксис, который отсутствует в других языках: вы можете также использовать ключевое слово `Handles` с методом, подписанным на событие. Необходимым условием для этого является определение переменной с ключевым словом `WithEvents`:

```
Public Class Subscriber
Public WithEvents evd As EventDemo
Public Sub Handler(ByVal x As Integer) Handles evd.DemoEvent
  ' Реализация обработчика
End Sub
Public Sub Action()
  evd = New EventDemo()
  evd.FireEvent()
End Sub
End Class
```

Обобщения

Все три языка поддерживают создание и использование обобщений. Обобщения обсуждались в главе 9.

Для использования обобщений C# позаимствовал синтаксис из шаблонов C++ для определения обобщенных типов с угловыми скобками. C++/CLI использует тот же синтаксис. В Visual Basic обобщенный тип определен с ключевым словом `Of` в скобках.

```
// C#
List<int> intList = new List<int>();
intList.Add(1);
intList.Add(2);
intList.Add(3);
// C++/CLI
List<int>^ intList = gcnew List<int>();
intList->Add(1);
intList->Add(2);
intList->Add(3);
```

```
' Visual Basic
Dim intList As List(Of Integer) = New List(Of Integer)()
intList.Add(1)
intList.Add(2)
intList.Add(3)
```

Поскольку вы используете угловые скобки в объявлении класса, компилятор знает, что он должен создать обобщенный тип. В конструкции `where` определены ограничения (constraints).

```
public class MyGeneric<T>
   where T : IComparable<T>
{
  private List<T> list = new List<T>();
  public void Add(T item)
  {
     list.Add(item);
  }
  public void Sort()
  {
     list.Sort();
  }
}
```

Определение обобщенного типа в C++/CLI подобно определению шаблона в C++. Вместо ключевого слова `template` с обобщениями используется ключевое слово `generic`. Конструкция `where` похожа на такую же в C#; однако в C++/CLI не поддерживаются ограничения конструктора.

```
generic <typename T>
where T : IComparable<T>
ref class MyGeneric
{
private:
   List<T>^ list;
public:
  MyGeneric()
  {
       list = gcnew List<T>();
  }
  void Add(T item)
  {
       list->Add(item);
  }
  void Sort()
  {
       list->Sort();
  }
};
```

В Visual Basic обобщенный класс определяется с помощью ключевого слова `Of`. Ограничения могут быть определены с помощью ключевого слова `As`.

```
Public Class MyGeneric(Of T As IComparable(Of T))
Private myList = New List(Of T)
Public Sub Add(ByVal item As T)
  myList.Add(item)
End Sub
Public Sub Sort()
  myList.Sort()
End Sub
End Class
```

Запросы LINQ

Интегрированные в язык запросы — средство C# 3.0 и Visual Basic 9.0. Синтаксис, используемый в двух языках, очень похож.

LINQ обсуждается в главе 11.

```
// C#
var query = from r in racers
          where r.Country == "Brazil"
          orderby r.Wins descending
          select r;
' Visual Basic
Dim query = From r in racers _
          Where r.Country = "Brazil" _
          Order By r.Wins Descending _
          Select r
```

C++/CLI не поддерживает запросов LINQ.

Смешивание "родного" и управляемого кода в C++/CLI

Одним из наибольших преимуществ C++/CLI является возможность смешивания "родного" и управляемого кода. Использование родного кода из C# осуществляется через механизм, известный, как *вызов платформы* (platform invoke). Этот механизм обсуждается в главе 24. Использование "родного" кода из C++/CLI известно под термином *It just works* ("Это просто работает").

В управляемом классе вы можете использовать как родной, так и управляемый код, как видно в приведенном ниже фрагменте. То же справедливо и для родного класса. Вы также можете смешивать родной и управляемый код внутри метода.

```
#pragma once
#include <iostream> // включаем этот файл для cout
using namespace std; // заголовок iostream определен в пространстве имен std
using namespace System;
public ref class Managed
{
public:
  void MixNativeAndManaged()
  {
       cout << "Родной код" << endl;
       Console::WriteLine("Управляемый код");
  }
};
```

В управляемом классе вы также можете объявить поле родного типа или указатель на родной тип. Сделать то же самое другим способом напрямую невозможно. Вы должны позаботиться о том, чтобы экземпляр управляемого типа мог быть перемещен сборщиком мусора при очистке памяти.

Для использования управляемых классов как членов родных классов в C++/CLI определено ключевое слово `gcroot`, которое находится в заголовочном файле `gcroot.h`. `gcroot` является оболочкой для `GCHandle`, который отслеживает объект CLR из родной ссылки.

```
#pragma once
#include "gcroot.h"
using namespace System;
public ref class Managed
{
public:
  Managed() { }
  void Foo()
  {
    Console::WriteLine("Foo");
  }
};
public class Native
{
private:
  gcroot<Managed^> m_p;
public:
  Native()
  {
    m_p = gcnew Managed();
  }
  void Foo()
  {
    m_p->Foo();
  }
};
```

Специфика C#

Некоторые синтаксические средства C# не были рассмотрены в этом приложении. В C# определен оператор `yield`, который облегчает создание перечислителей. Данный оператор не доступен в C++/CLI и Visual Basic; в этих языках перечислители должны быть реализованы вручную. К тому же C# определяет специальный синтаксис для допускающих `null` типов, в то время как другие языки предполагают применение вместо них обобщенных структур `Nullable<T>`.

C# допускает блоки небезопасного кода, в которых вы можете использовать указатели и арифметику указателей. Это средство чрезвычайно полезно для вызова методов из "родных" библиотек. Visual Basic не имеет такой возможности; в этом и состоит реальное преимущество C#. C++/CLI не нуждается в ключевом слове `unsafe`, чтобы определять блоки небезопасного кода. Для C++/CLI очень естественно смешивать "родной" и управляемый код.

Резюме

В этой главе вы узнали, как отобразить синтаксис C# на Visual Basic и C++/CLI. C++/CLI определяет расширения C++ для написания приложений .NET и следует за C# в расширениях синтаксиса. Хотя C# и C++/CLI имеют общие корни, все же между ними есть много важных отличий. Visual Basic не использует фигурных скобок, но вместо этого более многословен.

На примерах сравнения синтаксиса вы увидели, как отобразить синтаксис C# на C++/CLI и Visual Basic; как выглядят другие языки и как в них определяются типы, методы, свойства; какие ключевые слова используются для объектно-ориентированных средств; как выполняется управление ресурсами; как реализованы в этих трех языках делегаты, события и обобщения.

Хотя большую часть синтаксиса можно отобразить, все же эти языки отличаются по своей функциональности.