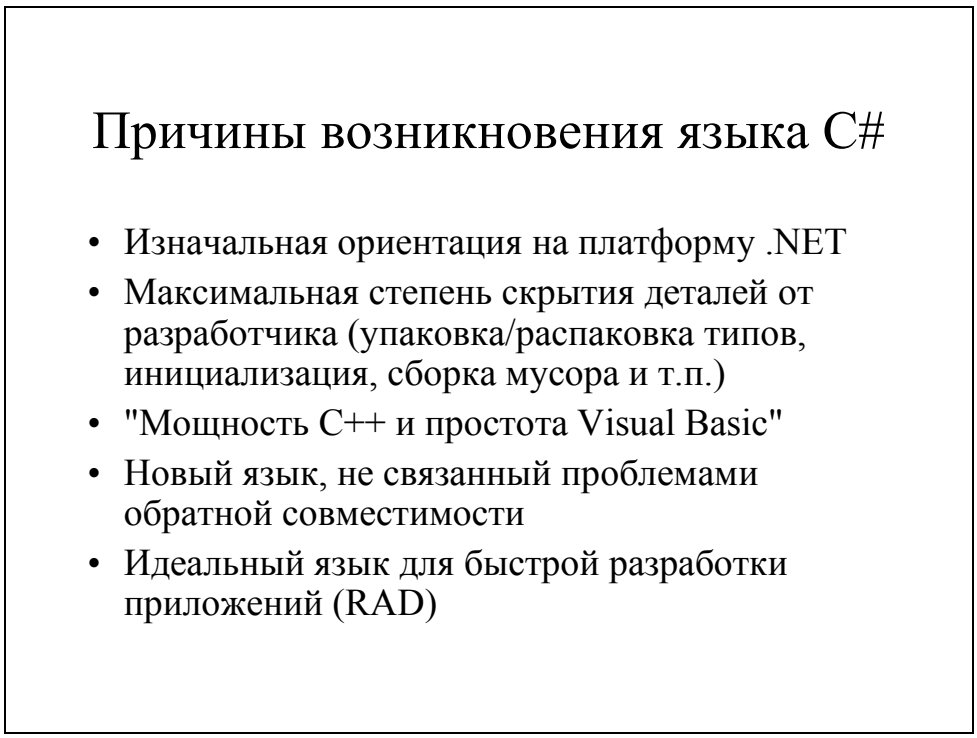
2. Обзор языка C#

- Основные принципы создания C#
- Базовые конструкции языка
- Препроцессор C#
- Системные и пользовательские атрибуты
- Опасный код в C#
- Описание языка Си-бемоль, являющегося подмножеством C#

Лекция 2. Обзор языка C#

В этой лекции рассматриваются следующие вопросы:

- Основные идеи создания C#
- Базовые конструкции языка
- Препроцессор C#
- Системные и пользовательские атрибуты
- Опасный код в C#
- Описание языка C♭, являющегося подмножеством C#



Причины возниковения языка C#

Главной особенностью языка C# является его ориентированность на платформу Microsoft .NET — создатели C# ставили своей целью предоставление разработчикам естественных средств доступа ко всем возможностям платформы .NET. Видимо, это решение можно считать более или менее вынужденным, так как платформа .NET изначально предлагала значительно большую функциональность, чем любой из существовавших на тот момент языков программирования.

Кроме того, создатели C# хотели скрыть от разработчика как можно больше незначительных технических деталей, включая операции по упаковке/распаковке типов, инициализации переменных и сборке мусора. Благодаря этому программист, пишущий на C#, может лучше сконцентрироваться на содержательной части задачи. В процессе решения этой задачи проектировщики C# пытались учесть уроки реализации Visual Basic'a, который достаточно успешен в скрытии деталей реализации, но недостаточно эффективен для написания крупных промышленных систем: создатели C# декларируют, что новый язык обладает мощностью C++ и в то же время простотой Visual Basic'a.

Еще одно преимущество создания нового языка программирования по сравнению с расширением существующих заключается в том, что при создании нового языка нет необходимости заботиться о проблемах обратной совместимости, которые обычно заметно затрудняют исправление застарелых проблем и даже внесение новых свойств в стандарт языка (подробное описание трудностей, возникающих при расширении старого языка программирования, можно прочитать в книге Б. Страуструпа "Дизайн и эволюция языка C++", М.: ДМК, 2000).

Таким образом, C# представляет собой новый язык программирования, ориентированный на разработку для платформы .NET и пригодный как для быстрого прототипирования приложений, так и для разработки крупномасштабных приложений.

Простота C#

- Многие языки программирования обладают запутанным синтаксисом, приводящим к трудностям как при компилировании программ, так и при их написании
- Создатели C# предпринимали специальные усилия для упрощения языка:
  - Запрет прямой манипуляции памятью
  - Более строгие правила преобразования типов
  - Отказ от провала в следующую ветку в switch
  - Запрещение множественного наследования

Простота C#

Многие существующие языки программирования обладают весьма запутанным синтаксисом и конструкциями с неочевидной семантикой — достаточно вспомнить сверхперегруженную значениями открывающую фигурную скобку в C++, использование ключевых слов в качестве идентификаторов в PL/I или проблемы отличия описателей видов от операций в Алголе 68. Все эти языковые особенности затрудняют написание компиляторов и служат источником труднонаходимых ошибок при создании программ. На другом полюсе этой проблемы находится язык Паскаль, в котором в целях упрощения было решено пожертвовать даже очевидно удобными для программиста свойствами.

C# занимает некоторую промежуточную позицию: из стандарта языка убраны наиболее неприятные и неоднозначные особенности C++, но в то же время язык сохранил мощные выразительные возможности, присущие для таких языков, как C++, Java или VB.

Укажем некоторые особенности языка C++, которые не поддерживаются C#:

- По умолчанию, C# запрещает прямое манипулирование памятью, предоставляя взамен богатую систему типов и сборку мусора. Непосредственная работа с памятью по-прежнему доступна в специальном режиме "опасного" кода (об этом см. ниже), но требует явного декларирования. Как следствие, в C# активно используется всего один оператор доступа "."
- Преобразования типов в C# значительно строже, чем в C++, в частности, большинство преобразований может быть совершено только явным образом. Кроме того, все приведения должны быть безопасными (т.е. запрещены неявные преобразования с переполнением, использование целых переменных как указателей и т.п.). Естественно, это заметно упрощает анализ типов при компиляции
- Одной из типичных ошибок в C++ было отсутствие оператора break при обработе одной из веток оператора switch. Проблема "провала" (fall-through) в C# решена кардинальным образом: компилятор требует наличия явного оператора перехода (break или goto case <name>) в любой ветке
- В C#, как и в Java, нет множественного наследования, вместо него предлагается использовать реализацию нескольких интерфейсов. Несмотря на то, что мнения по поводу множественного наследования сильно разнятся, отсутствие этого механизма в C# должно по крайней мере облегчить разработку компилятора

Управляющие конструкции C#

- C# содержит традиционный набор императивных конструкций:
  - Присваивания
  - Операторы ветвления (if-then-else, switch)
  - Циклы (do-while, for, foreach)
  - Исключения (try-catch-finally, throw)

Управляющие конструкции C#

Большинство современных языков программирования содержит приблизительно одинаковый набор конструкций управления и C# не предлагает в этой области ничего радикально нового.

В C# используются обычные присваивания для простых переменных; структурные или массовые присваивания не поддерживаются.

Операторы ветвления тоже достаточно традиционны (if, switch), но обладают двумя особенностями. Во-первых, условие в операторе if должно вырабатывать именно булевское значение (т.е. целого значения, вырабатываемого при присваивании недостаточно), а во-вторых, каждая ветка case внутри оператора switch должна содержать явное указание о дальнейшем потоке управления (т.е. либо break, либо goto на какую-то переменную, например, на метку другой ветки).

Что касается циклов, то C# поддерживает вполне традиционные циклы, такие как do-while, while-do и циклы с итерацией for, но помимо этого, поддерживает перебор массивов и коллекций с помощью оператора foreach, как в следующем примере:

```
Hashtable ziphash = new Hashtable();
...
foreach (string zip in ziphash.Keys)
{
   Console.WriteLine(zip + "          " + ziphash[zip]);
}
```

В этом примере надо обратить внимание на то, что перебираемый класс должен поддерживать интерфейс IEnumerator, а также на тот факт, что каждый извлекаемый из коллекции объект явным образом приводится к заявленному типу перебора (в нашем примере это string).

Наконец, C# поддерживает структурную обработку исключений с помощью конструкций try, catch и finally. Исключения можно генерировать и явным образом с помощью конструкции throw.

Типы-значения в C#

- Типы данных в C# разделяются на ссылочные типы и типы-значения
- К типам-значениям относятся:
  - Простые
    - целочисленные (int - 32 бита, long - 64 бита)
    - char (16 бит, Unicode)
    - типы с плавающей точкой (могут выдавать обычные значения, +0, -0, +∞, -∞, NaN)
    - decimal (128-бит, с точностью до 28 знаков)
  - Структурные
  - Перечислимые: enum Days { Mon=1, Tue … }, по умолчанию типа int

Типы данных

Система типов C# полностью отражает систему типов .NET и в целом достаточно типична для современных языков. Важная особенность этой системы типов заключается в явном разделении всех типов на типы-значения и ссылочные типы (мы уже обсуждали эти поняития в лекции 1).

К типам-значениям относится широкий набор примитивных типов данных, включая целые числа различной разрядности, типы с плавающей запятой различной точности, специальный тип decimal с фиксированной точностью, предназначенный для финансовых вычислений, а также символьный тип char, способный хранить символы в формате Unicode и потому удобный при разработке интернациональных приложений. Все целочисленные типы существуют в двух вариантах: знаковом и беззнаковом. Важной особенностью переменных с плавающей запятой является то, что операции над ними никогда не производят исключений, но зато результатом работы с ними могут быть значения ±∞ (как результат деления на ноль) или нечисловым значением, NaN (Not-A-Number, как результат деления 0 на 0 или операций, в котором один из операндов уже равняется NaN).

Отметим, что C# содержит также специальный тип для булевских значений; переменные булевского типа могут содержать значения true или false, но в отличие от большинства современных языков программирования этим значениям не соответствует никаких численных эквивалентов (в отличие, скажем, от VB, где False соответствует 0, а True почему-то соответствует значение -1).

Помимо примитивных типов, в C# существует возможность организовывать данные в структуры, состоящие из переменных любого типа, или в перечисления, составленные из нескольких переменных одного и того же целочисленного типа. Важной особенностью перечислений в C# является необходимость явного приведения к базовому типу при желании проинтерпретировать значение из перечисления как число.

Ссылочные типы

- Тип object (для упаковки значений)
- Тип class
- Интерфейсы
- Представители (delegates)
- Тип string
- Массивы

Ссылочные типы

Все ссылочные типы произведены от базового типа *object*, являющегося точным эквивалентом класса System.Object. Сам по себе тип object в основном используется для упаковки значений, подробно обсуждавшейся выше. Другая, более важная роль типа object заключается в том, что он является корнем для всей иерархии объектов в .NET и по умолчанию все *классы* унаследованы именно от System.Object. Классы могут иметь свои поля, методы, реализовывать интерфейсы и т.п. Так как классы представляют собой основной механизм организации программ и данных в C#, мы уделим им особое внимание в последующих слайдах.
Интерфейс представляет собой ссылочный тип, который может иметь только абстрактные элементы (аналогично методам, равным нулю, в C++). В частности, с помощью интерфейсов можно реализовать механизм множественного наследования — для этого необходимо унаследовать класс от нескольких элементов и затем явным образом реализовать заявленную функциональность.

Представители (delegates) являются относительно безопасной версией указателей на функции: окружение .NET гарантирует, что представители указывают именно на допустимый объект, а не просто на некоторый адрес в памяти. Основные области применения представителей — это методы обратного вызова и асинхронные обработчики событий.

Строки в C# являются полноценным ссылочным типом, но при этом обладают семантикой сравнений, характерной для типов-значений (т.е. строки равны, если равны их значения). Это можно проиллюстрировать следующим примером:

```
string str1 = "Hello World"; string str2 = "Hello " + "World";
if (str1 == str2)
   System.Console.WriteLine ("Strings are equal"); // will be executed
```

Массивы в C# бывают двух типов: многомерные (например, int [,,] определяет трехмерный массив) и невыровненные (они же массивы массивов, используемые в C++; например, int [] [] определяет двухмерный массив). Несмотря на простой внешний вид, массивы являются полноценными объектами, представляющими класс System.Array.

Классы

- Модификаторы доступа
- Конструкторы/деструкторы
- Методы объектов и их параметры
- Свойства (properties) и доступ к ним
- Индексаторы
- События (events) и представители (delegates)

Классы

Классы – это основной способ организации данных в C#; любая исполняемая программа, написанная на этом языке, должна представлять собой класс (так что C# представляет собой "настоящий" объектно-ориентированный язык, в отличие, скажем, от C++, в котором использование объектов возможно, но необязательно).

На следующих слайдах мы рассмотрим языковые возможности, связанные с классами:

- Различные модификаторы доступа, которые могут быть использованы в классах
- Конструкторы и деструкторы в классах
- Методы объектов и их параметры
- Свойства классов (properties) и способы доступа к ним
- Индексаторы
- События и представители

Модификаторы

- Модификаторы доступа
  - public, protected, private, internal
- Модификаторы элементов класса:
  - const, event, extern, override, readonly, static, virtual
- Модификаторы класса:
  - abstract, sealed

Модификаторы

Одним из важных принципов объектно-ориентированного подхода к программированию является инкапсуляция данных: считается, что внутреннее устройство класса и конкретная реализация его методов должны быть неизвестны внешним потребителям. Для выражения различных уровней доступности элементов программы в C# существует обширный набор модификаторов, позволяющий утверждать, что на данный момент C# обладает наиболее развитыми средствами поддержки инкапсуляции. Например, поля могут быть описаны следующим образом:

- public (данный элемент класса доступен всем внешним потребителям)
- protected (к такому элементу класса могут обращаться только классы, унаследованные от данного)
- private (элемент недоступен за пределами описания класса, т.е. недоступен даже потомкам данного класса; этот модификатор ставится по умолчанию)
- internal (элемент доступен только для классов, определенных в той же сборке, что и данный класс)

Кроме того, существуют модификаторы, изменяющие поведение элементов класса:

- const (свидетельствует, что данная переменная не может быть изменена)
- event (указывает, что данный элемент описывает событие; о событиях см. ниже)
- extern (обычно описывает метод с внешней реализацией, чаще всего, импортированный из Win32)
- override (используется в случае новой реализации виртуального метода)
- readonly (описывает переменные, которые могут получить значение только при самом описании или в конструкторе класса)
- static (указывает, что данный элемент принадлежит типу объекта, а не конкретному экземпляру)
- virtual (описывает метод, который может быть переопределен в потомках класса)

Наконец, класс в целом может иметь следующие дополнительные модификаторы:

- abstract (обозначает, что данный класс не может быть использован самостоятельно, а является только базой для классов-потомков)
- sealed (от таких классов нельзя наследовать; кстати, по очевидным соображениям комбинация модификаторов abstract sealed запрещена)

Конструкторы/деструкторы

```
class TestClass {
  public TestClass() { … }
  public ~TestClass () { ReleaseResources(); }
  public ReleaseResources() {
    // closing connections, releasing system resources etc.
  }
```

Конструкторы/деструкторы не возвращают значений.

Для статических классов конструкторы могут быть закрытыми (private).

Конструкторы и деструкторы

Конструкторы используются при создании конкретных экземпляров класса. Чаще всего, задачей конструктора является инициализация значений, используемых при дальнейшей работе с данным классом. Конструкторы не имеют возвращаемого значения. Если класс не содержит ни одного явного описания конструктора, то компилятор генерирует пустой конструктор, в котором выполняется единственное действие — вызов базового класса (если таковой существует). Обычно конструкторы объявляются с модификатором public, но возможно определение закрытого (private) конструктора, например, в случае класса без методов (такие классы иногда специально создаются для хранения статических или глобальных переменных, так как в C# любая переменная должна принадлежать какому-либо объекту). Приведем примеры конструкторов для класса Матрица:

```
public Matrix() // implements default constructor
{
   for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
         elements[i,j] = 0;
}

public Matrix (int val) // implements constructor with one parameter
{
   for (int i=0; i<n; i++)
      for (int j=0; j<n; j++)
         elements[i,j] = val;
}
```

В C# нет деструкторов в привычном понимании этого слова, т.е. объекту не может быть приписан специальный метод, физически уничтожающих объект по явному запросу пользователя. Дело в том, что за освобождение памяти в .NET отвечает механизм сборки мусора и потому явное уничтожение объектов в C# не предусмотрено. Предусмотрены только так называемые *завершители (finalizers)*, которые вызываются сборщиком мусора непосредственно перед уничтожением объекта. Но так как невозможно предсказать когда произойдет сборка мусора (и даже произойдет ли вообще), более надежным способом считается выделение завершающих действий в отдельный метод с именем Close или Dispose. Этот метод может вызывать как завершитель, так и сам пользователь.

Методы и их параметры

- Параметры методов:
  - входные параметры могут передаваться по ссылке (ref) или по значению
  - для возврата дополнительных значений можно использовать параметры типа out
- Перегрузка методов
  - по умолчанию методы не являются виртуальными!
  - ключевое слово override (базовый метод недоступен)
  - ключевое слово new (базовый метод скрывается, но доступен путем приведения к типу базового класса)

Методы и их параметры

Все активные действия программ на C# выполняются в методах классов. Естественно, эти методы могут получать на вход параметры и выдавать значения. При передаче параметров в C# необходимо явно указывать способ передачи — по значению или по ссылке; в последнем случае переменной должно предшествовать ключевое слово ref. Кроме того, создатели C# предусмотрели возможность для возвращения более чем одного значения из метода – для этого помимо явного возвращаемого значения метода, необходимо описать один или несколько параметров метода с ключевым словом out. Компилятор C# проверяет, что ref-параметры инициализируются перед вызовом метода, а также, что out-параметры получают значение до выхода из метода.

С точки зрения перегрузки методов, отличительной особенностью C# является то, что методы по умолчанию не являются виртуальными. Это сделано для того, чтобы избежать ошибок, связанных со случайным переопределением унаследованных функций. Кроме того, в C# есть два способа переопределения виртуального метода: при использовании ключевого слова override базовый метод становится недоступным, а при использовании ключевого слова new базовый метод все еще может быть вызван путем явного приведения к типу базового класса, как в следующем примере:

```
class BaseClass {
  public void TestMethod() {
    Console.WriteLine ("BaseClass.TestMethod()");
  }
}
class DerivedClass : BaseClass {
  new public void TestMethod() {
    Console.WriteLine ("DerivedClass.TestMethod()");
  }
}
...
DerivedClass test = new DerivedClass();
test.TestMethod(); // напечатает DerivedClass.TestMethod
((BaseClass)test).TestMethod(); // напечатает BaseClass.TestMethod
```

Свойства класса и доступ к ним

- Поля: `public int stateOfVeryCriticalResource;`
- Свойства:

```
private int m_stateOfVeryCriticalResource;
public int stateOfVeryCriticalResource {
  get { if (IsAllowedUser())
           return m_stateOfVeryCriticalResource; }
  set { if (IsAdmin())
           m_stateOfVeryCriticalResource = value; }
}
...
stateOfVeryCriticalResource = vcrCompletelyScrewedUp;
```

Свойства класса и доступ к ним

В объектно-ориентированном программировании считается "хорошим тоном" организовывать доступ к данным через специальные методы доступа get и set. Однако до недавнего времени эта рекомендация, к сожалению, совершенно не поддерживалась языками программирования. В C# такая языковая возможность наконец-то появилась. Теперь обычное описание поля можно дополнить методами доступа get и set и тогда при любом чтении поля или при присваивании ему значения будет обязательно выполняться функциональность, записанная в этих методах доступа (обратите внимание, что в методе set используется ключевое слово value).

Методы доступа очень удобны в тех случаях, когда необходимо проверить допустимость присваиваемого значения или достаточность полномочий запрашивающего приложения для доступа к данному полю. Первый случай можно проиллюстрировать таким фрагментом программы:

```
private int m_AgeOfClient;
public int AgeOfClient {
   get { if (AccessToPersonalInfoAllowed()) return m_AgeOfClient; }
   set { if (value > 0 && value <= 120)
            m_AgeOfClient = value;
         else
            MessageBox.Show("This client is not recommended for insurance");
       }
}
```

Второй случай возникает, например, в следующей программе:

```
private int m_stateOfVeryCriticalResource;
public int stateOfVeryCriticalResource {
  get { if (IsAllowedUser())
           return m_stateOfVeryCriticalResource; }
  set { if (IsAdmin())
           m_stateOfVeryCriticalResource = value; }
}
...
stateOfVeryCriticalResource = vcrCompletelyScrewedUp;
```

Индексаторы

- Индексатор – это метод доступа к элементам массивов:

```
public class Matrix
{
        public const int n = 10;
        public int[,] elements = new int[n,n];

        public int this[int i, int j]
        {
                get { return elements[i,j]; }
                set { elements[i,j] = value; }
        }
}
```

Индексаторы

Методы доступа в C# доступны не только для простых переменных, но и для элементов массивов. Пример описания индексатора показан на слайде. Из этого описания видно, что индексатор не может иметь произвольного имени; в данном примере мы использовали ключевое слово this как обозначение интерфейса, заданного по умолчанию. Если в классе реализовано несколько интерфейсов, то можно ввести и дополнительные индексаторы, обозначая их как InterfaceName.this. Наконец, заметим, что один и тот же массив можно индексировать с помощью переменных различных типов (например, используя int как ключ или string как имя для поиска в базе данных).

В качестве примера приведем класс, реализующий работу с квадратными матрицами. Понятно, что нам было бы привычнее иметь прямой доступ к элементам, т.е. писать просто A[i,j], а не A.elements[i,j]. Реализуется это следующим образом:

```
public class Matrix
{
       public const int n = 10;
       public int[,] elements = new int[n,n];

       public int this[int i, int j]
       {
              get { return elements[i,j]; }
              set { elements[i,j] = value; }
       }
}
```

При таком описании допустимо следующее использование:

```
Matrix a = new Matrix();
a[0,0] = 1; a[1,5] = 5;
Matrix b = new Matrix();
b[0,0] = -4; b[1,5] = 10;
```

События

- Событийно-ориентированный подход типичен для современных приложений (работа в сети, GUI-интерфейсы и т.д.)
- В C# используется модель "публикация и подписка":

```
public delegate void EventHandler (string strText);
...
evsrc.EventHandler += new EventHandler(CatchEvent);
```

События

Многие современные приложения требуют применения событийно-ориентированного подхода, например, это необходимо при создании распределенной системы, обменивающейся сообщениями, или при написании программ, использующих GUI-интерфейс. В C# используется модель "публикация/подписка" — класс публикует те события, которые он может инициировать, а другие классы могут подписаться на получение извещений о них. Для реализации этой модели используются *представители (delegates)*, которые выполняют в C# роль "безопасных указателей на функцию".

В следующем примере, иллюстрирующем использование представителей в C#, необходимо обратить внимание на фрагмент, в котором последовательно происходят подписка на событие, обработка события с помощью зарегистрированного обработчика и отказ от дальнейшей обработки события (см. оператор `evsrc.TextOut -= EventHandler`)

```
using System;
public delegate void EventHandler (string strText);
class EventSource {
  public event EventHandler TextOut;
  public void TriggerEvent() {
    if (TextOut != null) TextOut("Event triggered..."); }
}
class TestApp {
  public static void Main() {
    EventSource evsrc = new EventSource();
    evsrc.TextOut += new EventHandler(CatchEvent);  evsrc.TriggerEvent();
    evsrc.TextOut -= new EventHandler(CatchEvent);  evsrc.TriggerEvent();
    TestApp theApp = new TestApp();
    evsrc.TextOut += new EventHandler(theApp.InstanceCatch);
    evsrc.TriggerEvent();
  }
public static void CatchEvent(string strText) { WriteLine(strText); }
public void InstanceCatch(string strText) { WriteLine("Instance "+strText); }
```

Перегрузка операторов

- Перегрузка – один из наиболее спорных механизмов современных языков
- Перегрузка используется для сокращения записи операций над объектами

```
public static Matrix operator+
  (Matrix left, Matrix right)     { ... }
...
Matrix c = new Matrix();
c = a + b;
```

Перегрузка операторов

Перегрузка операторов — это один из наиболее спорных механизмов в современных языках. Некоторые программисты считают, что перегрузка операторов помогает только в создании трудно находимых ошибок. Другие программисты считают механизм перегрузки операторов полезным как раз из соображений улучшения читаемости кода. Как бы то ни было, перегрузка операторов стала неотъемлемой частью C#. Например, большинство классов в C# по умолчанию перегружают оператор сравнения (операция == практически всегда означает вызов метода Equals, унаследованного от System.Object).

Перегрузка операторов обычно используется для того, чтобы сократить и привести к привычному виду запись операций над объектами, определенными программистом. Скажем, с объектами, представляющими математические или физические величины, обычно ассоциируются арифметические операции. Возьмем, в качестве примера квадратные матрицы — для них естественно ввести операции сложения и умножения. Без перегрузки операторов эти действия пришлось бы записывать таким образом: `A.Add(B)` или `Matrices.Add (A,B)`. И та, и другая формы записи несколько непривычны, так как традиционной формой является запись вида A+B. Попробуем реализовать ее на C#:

```
public class Matrix {
   ...
   public static Matrix operator+ (Matrix left, Matrix right)
   {
      Matrix target = new Matrix();
      for (int i=0; i<n; i++)
         for (int j=0; j<n; j++)
            target[i,j] = left[i,j] + right[i,j];
      return target;
   }
```

При таком описании запись сложения матриц приобретает более знакомый вид:

```
Matrix c = new Matrix();
c = a + b;
```

Таким же образом можно было бы реализовать умножение и деление матриц, сравнение и другие операции.

Явные и неявные преобразования

- Приведения бывают явными и неявными
- Классы и структуры могут задавать применимые к ним явные и неявные приведения:

```
public struct Rational {
 public static implicit operator Rational(int i)
   { return new Rational(i,1); }
 public static explicit operator double (Rational r)
   { return (double) r.Numerator / r.Denominator }
```

Явные и неявные преобразования

Во всех языках допустимы присваивания, в которых участвуют переменные похожих, но все-таки различных типов. Например в C# допустимы следующие операторы:

```
short v1 = 44;
int v2 = v1;
```

Здесь во время второго присваивания выполняется *неявное приведение* переменной v1 к типу int. Однако неявные приведения возможны только при присваиваниях, в которых не может произойти потери данных, т.е. конечный тип должен содержать в себе все значения исходного типа[1]. Обратное преобразование должно сопровождаться *явным приведением*:

```
v1 = (short) v2;
```

Применим эту идею для организации сокращенной записи преобразований. Предположим, что у нас есть класс Rational, реализующий рациональные числа. Опишем набор приведений для этого класса (обратите внимание, что Rational является структурой, а не классом — структуры предоставляют практически все те же языкове возможности, что и классы):

```
public struct Rational {
  public int Numerator, Denominator;
  ...
  public static implicit operator Rational(int i)
    { return new Rational(i,1); }
  public static explicit operator double (Rational r)
    { return (double) r.Numerator / r.Denominator }
}

Rational r = 4; // implicit conversion
Rational r = new Rational(2,3);
double d = (double)r; // error without explicit conversion
```

[1] Отметим одно исключение: при преобразовании целой переменной в вещественный тип может произойти потеря точности, но с сохранением порядка

Список параметров

- В некоторых случаях точное количество параметров заранее неизвестно.
- В C# есть специальное ключевое слово params, которое позволяет задать целый список параметров
- params должен сопровождаться типом параметров (в общем случае – object) и быть последним параметром

Список параметров

В некоторых случаях точное количество параметров заранее неизвестно; например, метод вывода Console.WriteLine может принимать как одиночные переменные известного типа, так и произвольное число объектов для печати. В C++ для этого используется аргумент "многоточие"; в C# такая функциональность реализуется с помощью специального ключевого слова params, которое должно быть последним в списке параметров метода и сопровождаться указанием типа принимаемых параметров. Если в списке параметров могут встречаться аргументы различного типа, то необходимо использовать тип object, так как любой аргумент может быть приведен к этому типу путем упаковки.

Компилятор всегда начинает обработку вызова метода с попытки найти перегруженный метод, точно соответствущий типам переданных параметров, и только при отсутствии совпадающего метода обращается к варианту, использующему список параметров. Вызов со списком параметров обычно не очень эффективен, так как подразумевает создание и заполнение массива объектов, поэтому рекомендуется создавать различные перегруженные методы для наиболее употребительных случаев использования. Скажем, следующий пример с печатью трассирующих сообщений имеет смысл дополнить вариантами методов для одного, двух и трех параметров различных типов. Тогда вариант со списком параметров будет использоваться только в крайних случаях.

```
class Info {
  public static void Trace (string strMessage) {
    Console.WriteLine (strMessage);
  }
  public static void TraceX (string strFormat, params object[] list) {
    Console.WriteLine (strFormat, list);
  }
}

class Test {
  public static void Main() {
    Info.Trace  ("Plain vanilla string...");
    Info.TraceX ("{0} {1} {2}", "C", "U", 2001);
  }
}
```

Препроцессор C#

- Сбылась мечта Страуструпа: в C# препроцессор урезан (запрещены подстановки текста) и является частью компилятора
- Можно использовать #define, #undef для определения/отмены идентификаторов
- Механизм #if DEBUG… #else … #endif
- Генерация предупреждений и ошибок: #warning, #error

Препроцессор C#

В своей книге "Дизайн и эволюция языка C++" Бьярни Страуструп писал: "...Я поставил себе целью изжить Cpp [препроцессор C++]. Но задача оказалась труднее, чем представлялось вначале. Cpp, возможно, и неудачен, но трудно найти ему лучше структурированную и более эффективную замену". Программистам на C++ и сегодня приходится иметь дело с потенциально опасным механизмом препроцессирования, который может приводить к глобальным заменам во всей программе и от действия которого невозможно защититься структурным образом.

В C# эта проблема решена кардинальным образом. Препроцессор стал частью компилятора (т.е. перестал быть препроцессором в традиционном понимании этого слова), а из привычного набора макросов оставлены только следующие:

- #define и #undef для определения идентификаторов и отмены определения (но без значений — для задания значений используется ключевое слово const)
- Механизм условной компиляции, основанный на директивах #if, #else, #endif
- Генерация предупреждений и ошибок с помощью макросов #warning и #error

В следующем примере мы продемонстрируем использование всех перечисленных выше механизмов препроцессора:

```
#define DEBUG

#if DEBUG && DEMO
  #error You cannot build a debug demo version

class Demo {
  public static void Main() {
    #if DEBUG
      Console.WriteLine("Starting the program...");
    #endif
  }
}
```

Атрибуты

- Иногда бывает нужна информация времени проектирования/выполнения (требуется транзакция, событие по умолчанию и т.п.)
- Предлагаемое решение – использование атрибутов
- Атрибуты бывают системные (заранее определенные) и пользовательские

Атрибуты

При написании программ очень часто возникает потребность в записи какой-то важной информации о программе или ее отдельных компонентах. Например, программисту может понадобиться определить новые свойства для создаваемых им объектов или явно указать, требуется ли наличие транзакции. При этом невозможно заранее предугадать все возможные виды информации, которые могут потребоваться, и потому язык должен предоставлять программисту возможность создания новых типов информации, привязки их к объектам программы и средства для работы с ними. Традиционным решением этой задачи была запись информации подобного рода в специальных файлах (например, .IDL или .DEF). В C# для этой цели используются *атрибуты*, которые представляют собой "примечания" к элементам исходного текста программы (классам, методам, параметрам методов и т.д.). В отличие от комментариев, информация, записанная в атрибутах, не теряется во время компиляции, а сохраняется в метаданных программы и может быть извлечена с помощью механизма *рефлексии* (подробнее об этом – в лекции 14).

Существует целый ряд *системных атрибутов*, отражающих наиболее распространенные случаи использования сторонней информации. Приведем такой пример:

```
using System.Runtime.InteropServices;
public class AppMain {
  [DllImport("user32.dll")]
  public static extern int MessageBoxA(int handle, string message,
    string caption,int flags);
  public static void Main() {
    MessageBoxA(0,"Hello World","Native Message Box",0);
  }
  [conditional("DEBUG")]
  public static void SayHello() {
    Console.WriteLine("Hello, World!"); return;
  }
}
```

В этом примере используется атрибут DllImport, с помощью которого обеспечивается взаимодействие с функцией MessageBoxA из Win32 API, а затем используется атрибут conditional, который эквивалентен записи #if DEBUG. Если идентификатор DEBUG не определен, то и определения, и вызовы метода SayHello будут удаляться.

Пользовательские атрибуты

- Можно определять пользовательские атрибуты путем создания классов, наследующих от System.Attribute
- Пользовательские атрибуты доступны во время исполнения программы с помощью механизма рефлексии

Пользовательские атрибуты

Естественно, что программист может определять и собственные, *пользовательские атрибуты*. Для этого достаточно создать класс, унаследованный от System.Attribute. В следующем примере мы покажем создание и использование атрибута, задающего гиперссылку на страничку с информацией о классе и его методах:

```
public class HelpUrlAttribute : System.Attribute
{
    public HelpUrlAttribute(string url) { … }
    public string Url   { get {…} }
    public string Tag  { get {…}  set {…}   }
}
[HelpUrl("http://SomeUrl/MyClass")]
class MyClass  {}
[HelpUrl("http://SomeUrl/MyClass", Tag="ctor")]
class MyClass  {}
```

Информация, записанная в пользовательских атрибутах, может быть использована во время исполнения программы с помощью механизма рефлексии (см. следующий пример).

```
Type type = typeof(MyClass);
foreach(object attr in type.GetCustomAttributes() )
{
    if ( attr is HelpUrlAttribute ) {
        HelpUrlAttribute ha = (HelpUrlAttribute) attr;
        myBrowser.Navigate( ha.Url );
    }
}
```

Еще одним перспективным направлением для использования атрибутов является аспектно-ориентированное программирование — подход, в котором программист специально описывает степень участия каждой компоненты в общей системе в целях упрощения последующих изменений при сопровождении (подробнее об этом см. доклад Gregor Kiczales на ECOOP'99 Workshop on Aspect-Oriented Programming или страницу, посвященную этой методологии: http://aosd.net/)

Опасный код в C#

- В некоторых случаях программисту нужен полный контроль над памятью и прочими ресурсами приложения
- Для этого в C# предусмотрен режим опасного (unsafe) кода
- Чтобы избежать ошибок при сборке мусора, необходимо зафиксировать указатели, с которыми работает опасный код

Опасный код в C#

В общем и целом, C# достаточно успешно справляется с задачей скрытия излишних сложностей от программиста. Тем не менее, бывают случаи, когда программисту требуется полная свобода действий — например, речь может идти об ускорении каких-то фрагментов исходной программы или о необходимости работы со структурами, описанными в других языках и использующих указатели.

Специально для таких ситуаций в C# предусмотрена возможность написания *опасного (unsafe)* кода — для этого необходимо пометить метод или блок ключевым словом unsafe. Внутри опасных блоков можно применять операторы * и &, указатели, адресную арифметику и т.д., но, естественно, сгенерированная таким образом программа не будет гарантированно безопасной.

Еще одна особенность опасного кода — это возможность описания *фиксированных (fixed) указателей*. Дело в том, что прямой работе с указателями может помешать сборка мусора .NET, так как во время сборки мусора возможно перемещение всех объектов в куче. Понятно, что при перемещении объекта, с которым мы работаем через указатель, не может произойти ничего хорошего. Поэтому необходимо зафиксировать все указатели, с которыми работает опасный код; зафиксированные объекты сборщик мусора игнорирует.

Приведем пример использования опасного кода, который содержит строчку с целым строем звездочек:

```
class UnsafeTest
{
   unsafe static void SquarePtrParam (int* p) {
      *p *= *p;
   }

   unsafe public static void Main() {
      int i = 5;
      SquarePtrParam (&i);
      Console.WriteLine (i);
   }
}
```

Еще о C#

- В этом кратком изложении многие особенности C# остались за кадром (исключения, контроль версий, работа с переполнением, взаимодействие с другими языками и т.д.)
- C# наверняка будет развиваться и дальше; в момент написания курса обсуждался вопрос о включении в C# поддержки параметрического полиморфизма (generics)

Еще о C#

К сожалению, в этом кратком изложении очень многие особенности языка C# остались за кадром. Перечислим некоторые языковые механизмы, которые не были освещены в этой лекции:

- Исключения (впрочем, достаточно традиционные для современных языков: try-catch-finally)
- Встроенный механизм контроля версий (задача этого механизма – добиться, чтобы пользователям пришлось заменять или перекомпилировать старые библиотеки только в тех случаях, когда это действительно необходимо)
- Возможность отключения контроля переполнения (ключевое слово unchecked)
- Вопросы взаимодействия с другими языками (например, межъязыковая разработка и отладка)

Кроме того, можно с уверенностью говорить, что C# будет развиваться и дальше, так как все языки программирования продолжают развиваться и C# находится на переднем крае современного языкотворчества. Например, в момент написании данного курса широко обсуждался вопрос о включении в C# поддержки параметрического полиморфизма (generics). Подобный механизм мог бы существенно расширить выразительные возможности, доступные программистам на C#.

Си-бемоль

- Для того, чтобы научиться написанию компиляторов, необходима практика
- Наш курс будет сопровождаться написанием компилятора с небольшого языка, основанного на C#, и получившего название Си-бемоль ☺
- Этот язык содержит базовые конструкции языка C#, но с некоторыми ограничениями, упрощающими написание компиляторов
- Подробная спецификация этого языка приведена в приложении к курсу

C♭

Для того, чтобы научиться написанию компиляторов, необходима практика. Однако, реальные языки программирования не очень хорошо подходят в качестве тренировочного полигона, так как при их реализации чаще всего приходится существенно усложнять структуру компилятора из-за всяких второстепенных особенностей языка.

Поэтому в качестве примера к нашему курсу мы разработали специальный язык, полученный путем усечения C#. Этот язык получил название C♭, здесь мы вкратце опишем основные ограничения этого языка по сравнению с C# (более подробная спецификация этого языка приведена в приложении к курсу).

- Из типов данных поддержаны только следующие:
  - встроенные типы int, char, float, bool, string, void
  - классы (могут содержать поля и методы, но только вида public, private или static; вложенные классы запрещены)
  - интерфейсы (однако поддерживается только импорт существующих интерфейсов в целях совместимости с платформой .NET – описание новых интерфейсов запрещено)
  - одномерные массивы
  - тип данных "множество" (set), подобно одноименному типу в Паскале (этот тип введен в состав языка для того, чтобы продемонстрировать приемы реализации типов, отсутствующих в .NET)
- Из управляющих конструкций допускаются только if-then-else, while-do, do-while, присваивания, вызовы, пустой и составной операторы. Каждый оператор должен выдавать значение.
- Поддерживаются стандартные операции (+, –, *, /, %), причем тип операции всегда определяется по типу первого операнда.
- Перегрузка методов допускается, но при вызове список параметров должен определять вызываемый метод единственным образом.
- Все приведения должны быть записаны явным образом.

Курс сопровождается примером реализации C♭, который может быть использован как дополнительный демонстрационный материал во время лекций.

Литература к лекции

- Т. Арчер "Основы C#", Русская редакция, 2001. 448 с.
- Э. Гуннерсон "Введение в C#", СПб.: Питер, 2001. 304 с.
- "Microsoft C# Language Specification", Microsoft Press, 2001. 412 p.
- J. Trupin "Sharp New Language. C# Offers the Power of C++ and Simplicity of Visual Basic", MSDN Magazine, September 2000.

Литература к лекции

- Т. Арчер "Основы C#", Русская редакция, 2001. 448 с.
- Э. Гуннерсон "Введение в C#", СПб.: Питер, 2001. 304 с.
- "Microsoft C# Language Specification", Microsoft Press, 2001. 412 p.
- J. Trupin "Sharp New Language. C# Offers the Power of C++ and Simplicity of Visual Basic", MSDN Magazine, September 2000.