Класс может содержать объявления следующих членов:

- Конструкторы
- Деструкторы
- Константы — неизменные значения, известные во время компиляции и неизменяемые на протяжении времени существования программы. Константы объявляются с модификатором **const**.
- Поля — Поле - это переменная любого типа, которая объявлена непосредственно в классе или структуре.
- Методы
- Свойства
- Индексаторы
- Операторы
- События
- Делегаты
- Классы
- Интерфейсы
- Структуры

Конструкторы

Каждый раз, когда создается класс или структура, вызывается **конструктор**. Класс или структура может иметь несколько конструкторов, принимающих различные аргументы.

Конструкторы вызываются при создании экземпляров объекта с помощью оператора **new**

```
public class Taxi
{
    public bool isInitialized;
    public Taxi() { isInitialized = true; }   // Конструктор
 }

class TestTaxi
{
  static void Main()
   { Taxi t = new Taxi();
     Console.WriteLine(t.isInitialized);
   }
}
```

Этот конструктор экземпляра вызывается каждый раз при создании объекта на базе класса. Такой конструктор без аргументов называется конструктором по умолчанию.

Если класс не содержит конструктор, автоматически создается конструктор по умолчанию и для инициализации полей объекта используются значения по умолчанию.

Например, **int** инициализируется значением 0.

Эти значения можно посмотреть в Таблице.

Но если конструктор явно определен в классе, то конструктор по умолчанию использоваться не будет.

Тип значения	Значение по умолчанию
bool	**false**
byte	0
char	'\0'
decimal	0,0M
double	0,0D
enum	Значение, созданное выражением (E)0, где E — идентификатор перечисления.
float	0,0F
int (Целочисленное значение)	0
long	0L
sbyte	0
short	0
struct	Значение, полученное путем установки значений по умолчанию для полей типов значений и установки значения **null** для полей ссылочных типов.
uint	0
ulong	0
ushort	0

В этом примере класс **Person** не имеет конструкторов, поэтому автоматически предоставляется конструктор по умолчанию, а все поля инициализируются со значениями по умолчанию.

```
public class Person
{
  public int age;
  public string name;
}

class TestPerson
 {
  static void Main()
   {
    Person person = new Person();
    Console.WriteLine("Name: {0}, Age: {1}", person.name, person.age);
    Console.WriteLine("Press any key to exit."); Console.ReadKey();
    }
  }

// Output: Name: , Age: 0
```

При наследовании конструкторы и деструкторы не наследуются.

Передача управления конструктору базового класса осуществляется посредством конструкции

```
:base(...)
```

После ключевого слова **base** в скобках располагается список значений параметров конструктора базового класса.

В следующем примере демонстрируется использование инициализатора базового класса.

Класс **Circle** является производным от основного класса **Shape**, а класс **Cylinder** является производным от класса **Circle**.

Конструктор каждого производного класса использует инициализатор своего базового класса.

```
abstract class Shape
{
  public const double pi = Math.PI;
  protected double x, y;
  public Shape(double x, double y) { this.x = x; this.y = y; }
  public abstract double Area();
}
class Circle : Shape
{
   public Circle(double radius) : base(radius, 0) { }
   public override double Area() { return pi * x * x; }
}
class Cylinder : Circle
{
  public Cylinder(double radius, double height) : base(radius) { y = height; }
  public override double Area() { return (2 * base.Area()) + (2 * pi * x * y); }
}

class TestShapes
{
   static void Main()
     {
        double radius = 2.5;
        double height = 3.0;
        Circle ring = new Circle(radius);
        Cylinder tube = new Cylinder(radius, height);
        Console.WriteLine("Area of the circle = {0:F2}", ring.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());
        Console.WriteLine("Press any key to exit."); Console.ReadKey();
     }
 }

/*
   Output: Area of the circle = 19.63
   Area of the cylinder = 86.39
*/
```

Статические классы и структуры также могут иметь конструкторы.

Статический конструктор используется для инициализации любых статических данных или для выполнения определенного действия, которое требуется выполнить только один раз.

Он вызывается автоматически перед созданием первого экземпляра или ссылкой на какие-либо статические члены.

Статический конструктор не принимает модификаторы доступа и не имеет параметров. Статический конструктор нельзя вызывать напрямую.

```
class SimpleClass
{
   static readonly long baseline;
   static SimpleClass() { baseline = DateTime.Now.Ticks; }
}
```

Деструкторы

Деструкторы используются для уничтожения экземпляров классов.

- В структурах определение деструкторов невозможно. Они применяются только в классах.
- Класс может иметь только один деструктор.
- Деструкторы не могут наследоваться или перегружаться.
- Деструкторы невозможно вызвать. Они запускаются автоматически.
- Деструктор не принимает модификаторы и не имеет параметров.

```
class Car
{
   ~Car() // destructor
     {
         // cleanup statements...
     }
}
```

Деструктор неявным образом вызывает метод **Finalize** для базового класса объекта. Следовательно, предыдущий код деструктора неявным образом преобразуется в следующий код:

```
protected override void Finalize()
{ try
     {
       // Cleanup statements...
     }
finally
     {
       base.Finalize();
      }
}
```

Это означает, что метод **Finalize** вызывается рекурсивно для всех экземпляров цепочки наследования начиная с самого дальнего и заканчивая самым первым.

Пустые деструкторы использовать не следует. Если класс содержит деструктор, то в очереди метода **Finalize** создается запись. При вызове деструктора вызывается сборщик мусора, выполняющий обработку очереди. Если деструктор пустой, это приводит только к ненужному снижению производительности.

Программист не может управлять моментом вызова деструктора, потому что момент вызова определяется сборщиком мусора.

Сборщик мусора проверяет наличие объектов, которые больше не используются приложением. Если он считает, что какой-либо объект требует уничтожения, то он вызывает деструктор (при наличии) и освобождает память, используемую для хранения этого объекта.

Деструкторы также вызываются при выходе из программы.

Существует возможность принудительно выполнить сборку мусора, вызвав метод **Collect**, но в большинстве случаев этого следует избегать, потому что это может привести к проблемам с производительностью.

В целом, язык C# не требует управления памятью в той степени, в какой это требуется в случае разработки кода на C++. Это связано с тем, что сборщик мусора платформы .NET Framework неявным образом управляет выделением и высвобождением памяти для объектов.

Если объект требует уничтожения, то сборщик мусора запускает выполнение метода **Finalize** этого объекта.

Однако при инкапсуляции приложением неуправляемых ресурсов, таких как сетевые подключения, для высвобождения этих ресурсов следует использовать деструкторы.

В случае, когда приложением используется ценный внешний ресурс, также рекомендуется обеспечить способ высвобождения этого ресурса явным образом, прежде чем сборщик мусора освободит этот объект.

Это выполняется путем реализации метода **Dispose** интерфейса IDisposable, который выполняет необходимую очистку для объекта. Это может значительно повысить производительность приложения.

В следующем примере создаются три класса, образующих цепочку наследования. Класс **First** является базовым, класс **Second** является производным от класса **First**, а класс **Third** является производным от класса **Second**.

Все три класса имеют деструкторы.

В методе **Main( )** создается экземпляр самого дальнего в цепочке наследования класса.

При выполнении программы обратите внимание, что происходит автоматический вызов деструкторов всех трех классов по порядку от самого дальнего до первого в цепочке наследования.

```
class First
{
  ~First() { System.Diagnostics.Trace.WriteLine("First's destructor is called."); }
}

class Second : First
{
  ~Second() { System.Diagnostics.Trace.WriteLine("Second's destructor is called."); }
}

class Third : Second
{
 ~Third() { System.Diagnostics.Trace.WriteLine("Third's destructor is called."); }
}

Class TestDestructors
{
  static void Main()
    {
       Third t = new Third();
    }
 }
/* Output (to VS Output Window):
Third's destructor is called.
Second's destructor is called.
First's destructor is called.
*/
```

Методы

Метод представляет собой блок кода, содержащий набор инструкций.

Программа инициирует выполнение операторов, вызывая метод и задавая необходимые аргументы метода.

В C# все инструкции выполняются в контексте метода. Метод Main( ) является точкой входа для каждого приложения C#, и вызывается он средой **CLR** при запуске программы.

Методы объявляются в классе или в структуре путем указания уровня доступа, например public или private, необязательных модификаторов, например abstract или sealed, возвращаемого значения, имени метода и списка параметров этого метода. Все вместе эти элементы образуют сигнатуру метода. Следующий класс содержит два метода.

```
public class Motorcycle
{
   public void StartEngine()              {/* Method statements here */ }
   protected void AddGas(int gallons) { /* Method statements here */ }
}
```

В следующем примере определяется открытый класс, содержащий одно поле, метод и два конструктора.

```
public class Person
{
      public string name;                          // Поле

       public Person()                             // Конструктор без аргументов
       {
          name = "unknown";
        }
       public Person(string nm)                     // Конструктор с одним аргументом
       {
            name = nm;
        }
       public void SetName(string newName) // Метод
       {
          name = newName;
        }
}
```

Доступ к методам

Вызов метода объекта очень похож на обращение к полю. После имени объекта ставится точка, затем имя метода и скобки. В скобках перечисляются аргументы, разделенные запятыми.

```
class TestPerson
{
    static void Main()
    {
        Person person1 = new Person();                  // Конструктор без параметров.
        Console.WriteLine(person1.name);                // Обращение

        Person person2 = new Person("Sarah Jones");     // Конструктор с параметром.
        Console.WriteLine(person2.name);

        person1.SetName("John Smith");                  // Вызов метода
        Console.WriteLine(person1.name);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// John Smith
// Sarah Jones
```

Возвращаемые значения

Методы могут возвращать значения вызывающим их объектам. Если тип возвращаемого значения, указываемый перед именем метода, не равен void, для возвращения значения используется ключевое слово return.

В результате выполнения инструкции с ключевым словом return, после которого указано значение нужного типа, вызвавшему метод объекту будет возвращено это значение. Кроме того, ключевое слово return останавливает выполнение метода.

Если тип возвращаемого значения void, инструкцию return без значения все равно можно использовать для завершения выполнения метода. Если ключевое слово return отсутствует, выполнение метода завершится, когда будет достигнут конец его блока кода.

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
                { return number1 + number2; }

    public int SquareANumber(int number)
                { return number * number; }
}
```

Передача параметров по ссылке и по значению

По умолчанию при передаче методу по значению передается копия объекта, а не сам объект. Поэтому изменения в методе не оказывают влияния на исходную копию.

Когда объект ссылочного типа передается методу, передается ссылка на этот объект.

Создадим ссылочный тип с помощью ключевого слова **class**, как показано в следующем примере.

```
public class SampleRefType
 {
   public int value;
 }
```

```
public static void TestRefType()
{
  SampleRefType rt = new SampleRefType();
  rt.value = 44;
  ModifyObject(rt);
  Console.WriteLine(rt.value);
}

static void ModifyObject(SampleRefType obj)
{
  obj.value = 33;
}
```

В примере выполняются передача аргумента ссылочного типа. Метод **TestRefType** напечатает 33.

Для передачи по значению ссылочного типа можно использовать ключевое слово **ref**.

```
private void Calc(ref int Number)
{
    Number = 10;
}
int n = 1;
Calc(ref n);
Console.WriteLine(n);
```

В этом случае на экране появится число 10: изменение значения в методе сказалось и на главной программе. Такая передача метода называется *передачей по ссылке*, т.е. передаётся уже не копия, а ссылка на реальную переменную в памяти.

Если метод использует переменные по ссылке только для возврата значений и ему не важно что в них было изначально, то можно не инициализировать такие переменные, а передавать их с ключевым словом out. Компилятор понимает, что начальное значение переменной не важно и не ругается на отсутствие инициализации:

```
private void Calc(out int Number)
{
    Number = 10;
}
int n; // Ничего не присваиваем!
Calc(out n);
```

Параметры по умолчанию

Язык C# начиная с версии 4.0 (Visual Studio 2010) позволяет задавать некоторым параметрам значения по умолчанию – так, чтобы при вызове метода можно было опускать часть параметров. Для этого при реализации метода нужным параметрам следует присвоить значение прямо в списке параметров:

```
private void GetData(int Number, int Optional = 5)
{
    Console.WriteLine("Number: {0}", Number);
    Console.WriteLine("Optional: {0}", Optional);
}
```

В этом случае вызывать метод можно следующим образом:

```
GetData(10, 20);
GetData(10);
```

В первом случае параметр Optional будет равен 20, так как он явно задан, а во втором будет равен 5, т.к. явно он не задан и компилятор берёт значение по умолчанию.

Параметры по умолчанию можно ставить только в правой части списка параметров, например, такая сигнатура метода компилятором принята не будет:

```
private void GetData(int Optional = 5, int Number)
```

Перегрузка методов

Язык C# позволяет создавать несколько методов с одинаковыми именами, но разными параметрами. Компилятор автоматически подберёт наиболее подходящий метод при построении программы. Например, можно написать два отдельных метода возведения числа в степень: для целых чисел будет применяться один алгоритм, а для вещественных – другой:

```
/// <summary>
/// Вычисление X в степени Y для целых чисел
/// </summary>

private int Pow(int X, int Y)
{
    int b = 1;
    while (Y != 0)
        if (Y % 2 == 0)
        {
            Y /= 2;
            X *= X;
        }
        else
        {
            Y--;
            b *= X;
        }
    return b;
}
```

```
/// <summary>
/// Вычисление X в степени Y для вещественных чисел
/// </summary>

private double Pow(double X, double Y)
{
    if (X != 0)
            return Math.Exp(Y * Math.Log(Math.Abs(X)));
    else
            if (Y == 0)
                return 1;
            else
                return 0;
}
```

Вызывается такой код одинаково, разница лишь в параметрах – в первом случае компилятор вызовет метод Pow с целочисленными параметрами, а во втором – с вещественными:

```
Pow(3, 17);
Pow(3.0, 17.0);
```

Свойства (Properties)

- Конструкторы
- Деструкторы
- Константы
- Поля
- Методы
- Свойства
- Индексаторы
- Операторы
- События
- Делегаты
- Классы
- Интерфейсы
- Структуры

Свойства объединяют функции полей и методов.

Для объекта, использующего какой-либо объект, свойство является полем, поэтому для доступа к свойству требуется тот же синтаксис, что и для поля.

При реализации класса свойство является одним или двумя блоками кода, представляющими метод доступа get и/или метод доступа set.

Свойства можно использовать, как если бы они являлись открытыми членами данных, хотя в действительности они являются специальными методами, называемыми методами доступа.

Блок кода для метода доступа **get** выполняется, когда осуществляется чтение свойства;

Блок кода для метода доступа **set** выполняется, когда свойству присваивается новое значение. Свойство без метода доступа **set** считается доступным только для чтения.

Свойство без метода доступа **get** считается доступным только для записи.

Свойство с обоими методами доступа доступно для чтения и для записи.

Неявный параметр **value**, тип которого соответствует типу свойства используется для метода доступа **set**

```
class TimePeriod
{
   private double seconds;

   public double Hours
    {
       get { return seconds / 3600; }
       set { seconds = value * 3600; }
    }
}

class Program
{
    static void Main()
     {
         TimePeriod t = new TimePeriod();
         t.Hours = 24;                                          // Вызывается свойство set
         System.Console.WriteLine("Time in hours: " + t.Hours);    // Свойство get

      }
}

// Output: Time in hours: 24
```

В данном примере, класс TimePeriod хранит сведения о периоде времени в секундах, но свойство с именем Hours позволяет клиенту задать время в часах. Методы доступа для свойства Hours выполняют преобразование между часами и секундами.

Ключевое слово value используется для определения значения, присваиваемого методом доступа set

Статические поля и методы класса

У класса могут быть поля, связанные не с объектами, а с классом. Эти поля объявляются как статические с модификатором **static**.

Статические поля доступны всем методам класса. Независимо от объекта используется одно и то же статическое поле, позволяя использовать информацию, созданную другими объектами класса.

Аналогично полям у класса могут быть и статические методы, объявленные с модификатором **static**. Такие методы обрабатывают общую для класса информацию, хранящуюся в его статических полях.

Индексаторы (Indexers)

Индексаторы позволяют индексировать экземпляры класса или структуры так же, как массивы. Индексаторы напоминают свойства, но их методы доступа принимают параметры.

В следующем примере определяется универсальный класс и в качестве средств присвоения и извлечения значений создаются простые методы доступа get и set.

Класс Program создает экземпляр этого класса для хранения строк.

```
class SampleCollection<T>
{
   private T[ ] arr = new T[100];
   public T this[int i] // Определяет индексатор, который позволяет клиентскому коду использовать скобки [ ]
     {
        get { return arr[i];    }
        set { arr[i] = value;  }
      }
}


class Program
{
   static void Main(string[] args)
      {
          SampleCollection<string>  stringCollection = new SampleCollection<string>();
          stringCollection[0] = "Hello, World";                    // Используются [ ]
          System.Console.WriteLine(stringCollection[0]);
        }
}

// Output:
// Hello, World.
```

Перегруженные операции

Перегрузка операций строится на основе общедоступных (public) статических (вызываемых от имени класса) функций-членов с использованием ключевого слова **operator**.

+, –, !, ~, ++, —, true, false	Унарные символы операций, допускающие перегрузку true и false также являются операциями
+, –, *, /, %, &, |, ^, <<, >>	Бинарные символы операций, допускающие перегрузку
==, !=, <, >, <=, >=	Операции сравнения перегружаются
&&, ||	Условные логические операции моделируются с использованием ранее переопределенных операций & и |
[ ]	Операции доступа к элементам массивов моделируются за счет индексаторов
( )	Операции преобразования реализуются с использованием ключевых слов implicit и explicit
+=, –=, *=, /=, %=, &=, =, ^=, <<=, >>=	Операции не перегружаются, по причине невозможности перегрузки операции присвоения
=, ., ?:, –>, new, is, sizeof, typeof	Операции, не подлежащие перегрузке

При этом при перегрузке операций для любого нового типа выполняются следующие правила:

- префиксные операции ++ и — перегружаются парами;
- операции сравнения перегружаются парами: если перегружается операция ==, также должна перегружаться операция !=. То же самое относится к парам < и >, <= и >=.

Для перегрузки оператора в пользовательском классе нужно создать метод в классе с правильной сигнатурой.

Метод нужно назвать "**operator X**", где X — имя или символ перегружаемого оператора.

Унарные операторы имеют один параметр, а бинарные — два.

В каждом случае один параметр должен быть такого же типа, как класс или структура, объявившие оператор, как показано в следующем примере.

```
public static Complex operator +(Complex c1, Complex c2)
```

Унарные операторные функции

```
using System;
using System.Collections.Generic;
using System.Text;

namespace operations
{ // Для моделирования унарных постфиксных и префиксных операций может
  // потребоваться модификация структуры класса: объявляются дополнительные
  // переменные для сохранения значений координат.

public class Point2D
{  public float x, y;
   float xTmp, yTmp;
   public Point2D( ) { x = 0.0; y = 0.0; xTmp = 0.0; yTmp = 0.0; }
   public static Point2D operator ++(Point2D par)
      {      par.x = (par.xTmp)++;   // Фактическим координатам присваиваются старые значения.
             par.y = (par.yTmp)++;
             return par;
      }
    public static Point2D operator --(Point2D par)
       {    par.x = --(par.xTmp);    // Фактическим координатам присваиваются новые значения.
            par.y = --(par.yTmp);
            return par;
        }
 }
```

```
class Program
{
 static void Main(string[ ] args)
   {
      Point2D p = new Point2D( );
      int i;
      //  унарный плюс всегда постфиксный, а унарный минус всегда префиксный.

        for (i = 0; i < 10; i++) { p++; Console.WriteLine("{0:F3},{1:F3}",p.x, p.y); }
        Console.WriteLine("=========================================");

        for (i = 0; i < 10; i++) { ++p; Console.WriteLine("{0:F3},{1:F3}", p.x, p.y); }
        Console.WriteLine("=========================================");

        for (i = 0; i < 10; i++) { p--; Console.WriteLine("{0:F3},{1:F3}", p.x, p.y); }
         Console.WriteLine("=========================================");

          for (i = 0; i < 10; i++) { --p; Console.WriteLine("{0:F3},{1:F3}", p.x, p.y); }
           }
    }
}
```

Бинарные операции

Бинарные операции обязаны возвращать значения.

```
public static Point2D operator + (Point2D par1, Point2D par2)
  {    return new Point2D(par1.x+par2.x,par1.y+par2.y);   }
```

Реализуется алгоритм "сложения" значения типа Point2D со значением типа float. От перемены мест слагаемых сумма НЕ ИЗМЕНЯЕТСЯ. Однако эта особенность нашей операторной функции "сложения" (операции "сложения") должна быть прописана программистом. В результате получаем ПАРУ операторных функций, которые отличаются списками параметров.

```
// Point2D + float
public static Point2D operator + (Point2D par1, float val)
 { return new Point2D(par1.x+val,par1.y+val); }

// float + Point2D
public static Point2D operator + (float val, Point2D par1)
  { return new Point2D(val+par1.x,val+par1.y); }
```

А вот применение этих функций. Внешнее сходство выражений вызова операторных функций с обычными выражениями очевидно.

...p1 + p2... ...3.14 + p2... ...p2 + 3.14...

Операции сравнения реализуются по аналогичной схеме. Хотя не существует никаких ограничений на тип возвращаемого значения, в силу специфики применения (обычно в условных выражениях операторов управления) операций сравнения все же имеет смысл определять их как операторные функции, возвращающие значения true и false:

```
public static bool operator == (myPoint2D par1, myPoint2D par2)
{   if ((par1.x).Equals(par2.x) && (par1.y).Equals(par2.y))
            return true;
    else
          return false;
 }
public static bool operator != (myPoint2D par1, myPoint2D par2)
 {   if (!(par1.x).Equals(par2.x) || !(par1.y).Equals(par2.y))
            return true;
    else
           return false; }
```

Перегрузка булевских операторов true и false

В известном мультфильме о Винни Пухе и Пятачке, Винни делает заключение относительно НЕПРАВИЛЬНОСТИ пчел. Очевидно, что по его представлению объекты — представители ДАННОГО класса пчел НЕ удовлетворяют некоторому критерию.

В программе можно поинтересоваться непосредственно значением некоторого поля объекта:

```
Point2D p1 = new Point2D(GetVal(), GetVal());
.........
if ((p1.x).Equals(125)) {/*...*/} // Это логическое выражение!
```

Так почему же не спросить об этом у объекта напрямую? В классе может быть объявлена операция (операторная функция) true, которая возвращает значение true типа bool для обозначения факта true и возвращает false в противном случае

Классы, включающие объявления подобных операций (операторных функций), могут быть использованы в структуре операторов if, do, while, for в качестве условных выражений

При этом, если в классе была определена операция true, в том же классе должна быть объявлена операция false:

Перегрузка булевских операторов. Это ПАРНЫЕ операторы. Критерии ИСТИННОСТИ могут быть самые разные. В частности, степень удаления от точки с координатами (0,0).

```
public static bool operator true (Point2D par)
{
   if (Math.Sqrt(par.x*par.x + par.y*par.y) < 10.0)
         return true;
  else
        return false;
}

public static bool operator false (Point2D par)
{
   double r = (Math.Sqrt(par.x*par.x + par.y*par.y));
  if (r > 10.0 || r.Equals(10.0))
              return false;
       else
              return true;
}
```

Определение операций конъюнкция & и дизъюнкция |

После того как объявлены операторы true и false, могут быть объявлены операторные функции — конъюнкция и дизъюнкция. Эти функции работают по следующему принципу:

- после оценки истинности (критерий оценки задается при объявлении операций true и false) операндов конъюнкции или дизъюнкции функция возвращает ссылку на один из операндов либо на вновь создаваемый в функции объект;

- в соответствии с ранее объявленными операциями true и false, операторные конъюнкция и дизъюнкция возвращают логическое значение, соответствующее отобранному или вновь созданному объекту:

```
public static Point2D operator | (Point2D par1, Point2D par2)
{
    if (par1) return par1;                    // Определить "правильность" объекта par1 можем!
    if (par2) return par2;                    // Определить "правильность" объекта par2 можем!
    return new Point2D(10.0F, 10.0F); // Вернули ссылку на новый "неправильный" объект
}

public static Point2D operator & (Point2D par1, Point2D par2)
{
   if (par1 && par2)
        return par1;                    // Вернули ссылку на один из "правильных" объектов
   else
       return new Point2D(10.0F, 10.0F); // Вернули ссылку на новый "неправильный"
                                                 //  объект
}
```

Выражение вызова операторной функции "дизъюнкция" имеет вид

```
if (p0 | p1) Console.WriteLine("true!");
```

Определение операций || и &&

Эти операции сводятся к ранее объявленным операторным функциям. Обозначим символом T тип, в котором была объявлена данная операторная функция. Если при этом операнды операций && или || являются операндами типа T и для них были объявлены соответствующие операторные функции operator &( ) и/или operator |( ), то для успешной эмуляции операций && или || должны выполняться следующие условия:

- тип возвращаемого значения и типы каждого из параметров данной операторной функции должны быть типа T. Операторные функции operator & и operator | , определенные на множестве операндов типа T, должны возвращать результирующее значение типа T;

- к результирующему значению применяется объявленная в классе T операторная функция operator true (operator false).

При этом приобретают смысл операции && или ||. Их значение вычисляется в результате комбинации операторных функций operator true( ) или operator false() со следующими операторными функциями:

1. Операция x && y представляется в виде выражения, построенного на основе трехместной операции

```
T.false(x)? x: T.&(x, y),
```

где T.false(x) является выражением вызова объявленной в классе операторной функции false, а T.&(x, y) – выражением вызова объявленной в классе T операторной функции &.

Таким образом, сначала определяется "истинность" операнда x, и если значением соответствующей операторной функции является ложь, результатом операции оказывается значение, вычисленное для x.

В противном случае определяется "истинность" операнда y, и результирующее значение определяется как КОНЪЮНКЦИЯ истинностных значений операндов x и y.

2. Операция x || y представляется в виде выражения, построенного на основе трехместной операции

```
T.true(x)? x: T.|(x, y),
```

где T.true(x) является выражением вызова объявленной в классе операторной функции, а T.|(x, y) – выражением вызова объявленной в классе T операторной функции | .

Таким образом, сначала определяется "истинность" операнда x, и если значением соответствующей операторной функции является истина, результатом операции оказывается значение, вычисленное для x. В противном случае определяется "истинность" операнда y, и результирующее значение определяется как ДИЗЪЮНКЦИЯ истинностных значений операндов x и y.

При этом в обоих случаях "истинностное" значение x вычисляется один раз, а значение выражения, представленного операндом y, не вычисляется вообще либо определяется один раз.

Преобразования явные и неявные

explicit и implicit

Возвращаемся к проблеме преобразования значений одного типа к другому. И если для элементарных типов проблема преобразования решается (с известными ограничениями, связанными с "опасными" и "безопасными" преобразованиями), то для вновь объявляемых типов алгоритмы преобразования должны реализовываться разработчиками этих классов.

Логика построения явных и неявных преобразователей достаточно проста. Программист самостоятельно принимает решение относительно того:

- каковым должен быть алгоритм преобразования;
- будет ли этот алгоритм выполняться неявно или необходимо будет явным образом указывать на соответствующее преобразование.

Ниже рассматривается пример, содержащий объявления классов Point2D и Point3D. В классах предусмотрены алгоритмы преобразования значений от одного типа к другому, которые активизируются при выполнении операций присвоения:

Операторная функция, в которой реализуется алгоритм преобразования значения типа осуществляется с ЯВНЫМ указанием необходимости преобразования.

Принятие решения относительно присутствия в объявлении ключевого слова explicit вместо implicit оправдывается тем, что это преобразование сопровождается потерей информации.

Существует мнение, что об этом обстоятельстве программисту следует напоминать всякий раз, когда он в программном коде собирается применить данное преобразование.

```
using System;
 // Объявления классов.
 // Операторная функция, в которой реализуется алгоритм преобразования
 // значения типа Point2D в значение типа Point3D. Это преобразование
      осуществляется НЕЯВНО.

class Point3D
{
   public int x,y,z;
  public Point3D()  { x = 0; y = 0; z = 0; }
  public Point3D(int xKey, int yKey, int zKey) { x = xKey; y = yKey; z = zKey; }

   public static implicit operator Point3D(Point2D p2d)
    {
       Point3D p3d = new Point3D();
       p3d.x = p2d.x; p3d.y = p2d.y; p3d.z = 0;
       return p3d;
    }
 }
```

```
// Операторная функция, в которой реализуется алгоритм преобразования
// значения типа Point3D в значение типа Point2D. Это преобразование
// осуществляется с ЯВНЫМ указанием необходимости преобразования.

class Point2D
{
    public int x,y;
    public Point2D() { x = 0; y = 0; }
    public Point2D(int xKey, int yKey) { x = xKey; y = yKey; }

    public static explicit operator Point2D(Point3D p3d)
     {
        Point2D p2d = new Point2D(); p2d.x = p3d.x; p2d.y = p3d.y; return p2d;
     }
}
```

// Тестовый класс.

```
 class TestClass
{
    static void Main(string[ ] args)
        { Point2D p2d = new Point2D(125,125);
          Point3D p3d;                                  // Сейчас это только ссылка!
// Этой ссылке присваивается значение в результате
// НЕЯВНОГО преобразования значения типа Point2D к типу Point3D
        p3d = p2d;
 // Изменили значения полей объекта.
       p3d.x = p3d.x*2;
       p3d.y = p3d.y*2;
       p3d.z = 125;
// Главное – появилась новая информация, которая будет потеряна в случае присвоения
//  значения типа Point3D значению типа Point2D. Ключевое слово explicit в объявлении
// соответствующего метода преобразования вынуждает программиста подтверждать,
// что он в курсе возможных последствий этого преобразования.
      p2d = (Point2D)p3d;
      }
}
```