синтаксису C#
Часть I. Основы

Автор презентации: Бабин К.А., Архангельская область, город Новодвинск, учитель информатики и ИКТ в МОУ «СОШ №4» города Новодвинска

При создании данной презентации использовалась следующая печатная литература:

1. Г.Щилдт. C# 4.0: полное руководство. : Пер. с англ. - М.: ООО «И.Д. Вильямс», 2014. - 1056 с.: ил. - Парал. тит. англ.
2. Д.Албахари, Б.Албахари C# 7.0 “Справочник. Полное описание языка”, 7-е изд. : Пер. с англ. - СПб.: ООО «Диалектика», 2019. – 1024 с. : ил. – Парал. тит англ.
3. Э.Троелсен, Ф.Джепикс Язык программирования C# 7 и платформы .NET и NET.Core, 8-е изд. : Пер. с англ. — СПБ. : ООО «Диалектика», 2019 — 1328 с. : ил. — Парал. тит. Англ.

Использованные интегрированные среды разработки (IDE): Microsoft Visual C# 2010.

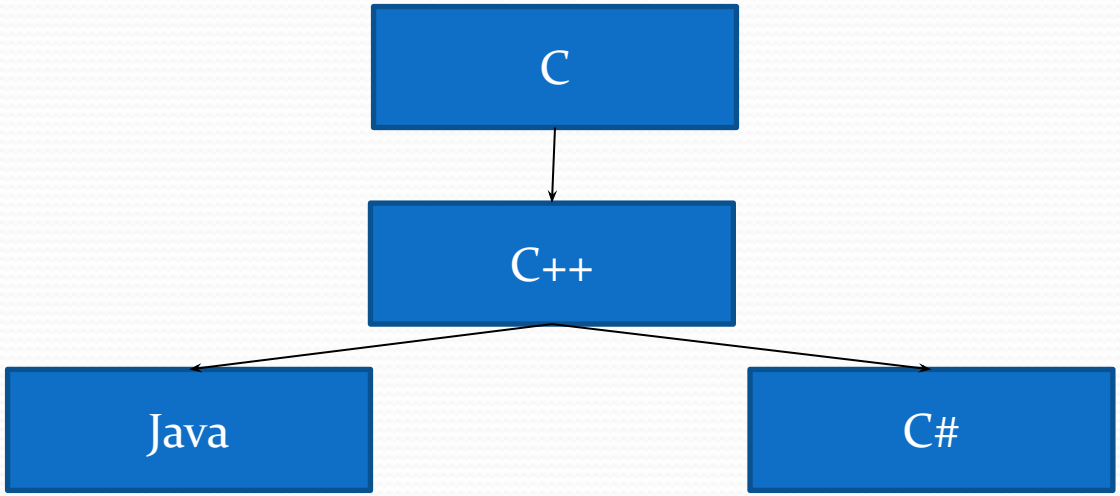
Новые средства C# 5.0-7.0 не рассматриваются в данной презентации (за некоторым исключением), т.к. не поддерживаются системой Yandex Contest при проверке олимпиадных заданий (На момент написания в Yandex Contest использовался компилятор Mono C# 5.2).

1. Общие сведения

1) Краткая информация
2) Основные правила языка C#
3) Шаблон для написания консольных программ
4) Альтернативный шаблон для написания консольных программ

Краткая информация

1) Язык программирования C# Разработан корпорацией Microsoft в конце 1990-х годов как часть общей стратегии .NET, т.е. у него имеется особая взаимосвязь со средой выполнения .NET Framework, предназначенной для разработки программ, в первую очередь, для операционной системы Windows;
2) Главный разработчик - Андерс Хейльсберг;
3) Является непосредственным наследником языков C и C++, используется так называемый «си-подобный» синтаксис.
4) Общеязыковая среда выполнения (CLR) как составная часть .NET Framework поддерживает многоязыковое программирование, а также обеспечивает переносимость и безопасное выполнение программ.
5) Наличие библиотеки классов. Эта библиотека предоставляет программе доступ к среде выполнения.
6) Любой создаваемый объект (как и в Java) - это класс, что подчеркивает объектно-ориентированную концепцию разработки программ.



Основные правила языка C#

1) Регистрозависимый. Например, нельзя использовать ключевые слова в регистре, в котором они не предусмотрены (например, INT или Int вместо int)

2) После каждой операции, кроме операций препроцессора и некоторых других необходимо ставить точку с запятой ; Точка с запятой означает конец оператора.

3) Для декорирования можно использовать сколько угодно табуляций и пробелов в коде, кроме целостных операций типа ==;

4) Переменная или функция может начинаться только с буквы, т. е. **1MyVar** – недопустимо. Также **не рекомендуется** начинать переменную с нижнего подчеркивания (типа _MyVar), это обычно зарезервировано для специфических нужд среды или используются стандартной библиотекой. В качестве имен нельзя использовать ключевые слова языка; Однако применение **@** перед именем снимает это ограничение, например, @var или @for (это отличие от других си-подобных языков), хотя это не рекомендуется.

5) Десятичные числа типа 0,54 записываются только через точку, т.е. 0.54 или .54

6) Константы в языке по общему соглашению принято писать заглавными буквами (const int MAX = 10);

7) Строго типизированный язык (не считая тип **dynamic**).

Шаблон для написания консольных программ

```
using System;
class Example
{
   static void Main()
   {
      // код программы
      Console.ReadLine();
   }
}
```

1) using System; делает видимым стандартное пространство имён;

Также совершенно не обязательно использовать стандартное пространство имён в глобальном блоке. Его можно использовать в том блоке, где это необходимо.

2) В C# всё является объектно-ориентированным, поэтому все глобальные блоки являются классами. Имя класса может быть любым, если не нарушается правило именования.

3) static void Main() – главная функция любой программы, её наличие обязательно.

4) Console.ReadLine(); – пауза после выполнения программы (ожидание нажатия Enter)

5) // это однострочный комментарий

/*это тоже комментарий, но многострочный*/

написания консольных программ

```
using System;
class Example
{
    static void Main()
    {
        // код программы

    System.Console.Read();
    }
}
```

Вместо using System; можно использовать System непосредственно перед операторами. Но чаще так не делают.

операции

1) Арифметические операции и операции сравнения
2) Другие операции
3) Сокращенные записи
4) Вывод на консоль

Арифметические операции и операции сравнения

Обозначение	Название операции
+	Сложение
-	Вычитание
*	Умножение
/	Деление (в том числе целочисленное, если оба значения (переменные) целочисленные)
%	Деление по модулю (взятие остатка от деления)
==	Равно
!=	Не равно
=	Присваивание
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
++	Инкремент (увеличение на 1), например i++ или ++i
--	Декремент (уменьшение на 1), например i-- или --i

Другие операции

оператор	Значение
!	НЕ (NOT в Pascal)
&	И (AND в Pascal) (Отличие от C++)
|	ИЛИ (OR в Pascal) (Отличие от C++)
^	Исключающее ИЛИ (XOR в Pascal) (Отличие от C++)
&&	Укороченное И (Отличие от C++)
||	Укороченное ИЛИ (Отличие от C++)
<<	Побитовый сдвиг влево (SHL в Pascal)
>>	Побитовый сдвиг вправо (SHR в Pascal)

Сокращенные записи

Сокращенная запись	Полная запись
`s += 5;`	`s = s + 5;`
`s -= 5;`	`s = s - 5;`
`s *= 5;`	`s = s * 5;`
`s /= 5;`	`s = s / 5;`
`s %= 5`	`s = s % 5;`
`S ^= 5`	`S = s ^ 5;` (Отличие от C++)
`S &= 5`	`S = s & 5;` (Отличие от C++)
`S |= 5`	`S = s | 5;` (Отличие от C++)

Вывод на консоль

Оператор	Пояснение
Console.Write(‘d’);	Вывод символа (одинарные кавычки)
Console.Write(sum);	Вывод переменной
Console.Write(“Hi, man!”);	Вывод строки (двойные кавычки)
Console.Write(“Here is ” + sum + “ dollars”;	Совмещение вывода строки / переменной / строки
Console.WriteLine();	Перевод на новую строку
Console.WriteLine(“Enter the ” + i + “-st number\n”);	Совмещение: строка / переменная / строка / перевод на новую строку
Console.Write(“Bye.\n”);	Тоже строка с переносом на новую строку
Console.WriteLine(“Here is {1} and {2} and {3}”, r, k, b)	Форматированный вывод (описывается позже)

3. Типы данных

1) Типы целочисленных переменных
2) Типы для представления чисел с плавающей точкой и десятичный тип данных
3) Символы и логический тип данных
4) Правила объявления переменных и область видимости
5) Формат записи при выводе
6) Суффиксы для обозначения типов констант
7) Коды управляющих последовательностей
8) Правила преобразования типов
9) Преобразование и приведение типов
10) Необходимое приведение типов
11) Математические функции (класс System.Math)
12) Использование математических функций (пример)

Типы целочисленных переменных

Переменная	Разрядность в битах	Диапазон представления чисел
Целочисленные (для 32-разрядной системы)		
byte	8	0 — 255
sbyte	8	-128 — 127
short	16	-32 768 — 32 767
ushort	16	0 — 65 535
int	32	-2 147 483 648 — 2 147 483 647
uint	32	0 — 4 294 967 295
long	64	-9 223 372 036 854 775 808 — 9 223 372 036 854 775 807
ulong	64	0 — 18 446 744 073 709 551 615

Типы для представления чисел с плавающей точкой и десятичный тип данных

Переменная	Разрядность в битах	Диапазон представления чисел
		Вещественные
float	32	5E-45 — 3,4E+38
double	64	5E-324 — 1,7E+308
decimal	128	1E-28 — 7,9E+28 Этот тип данных способен представлять **десятичные (в десятичной системе счисления)** значения без ошибок округления, он особенно удачен для расчетов, связанных с финансами, но вычисления выполняются примерно в 10 раз медленнее, чем с типом **double**, который оперирует **двоичными** данными. **Примечание: к значению переменной обязательно необходимо дописывать суффикс m, например 19.95m, иначе эти значения интерпретировались бы как стандартные константы с плавающей точкой с типом double, который несовместимы с типом данных decimal (без явного перевода).** Тем не менее этой переменной можно присвоить целое значение без суффикса, например, 10.

Символы и логический тип данных

В C# **символы** представлены не 8-разрядным кодом (как в некоторых других языках программирования), а 16-разрядным кодом, который называет Unicode.

Объявление символьной переменной:

```
char ch;
ch = ‘X’;
ch = 88; //это ошибка, так делать нельзя
ch = (char) 88; //А так можно, будет храниться номер символа из таблицы UTF под номером 88 (Это «X»)
```

Логический тип данных представляет два логических значения: true (“Истина”) и false (“Ложь”). Причем, в отличие от C++, присваивать переменным 1 или 0 вместо этих ключевых слов нельзя.

Объявление переменной логического типа:

```
bool b;
b = true;
b = a > b; (при известных значениях переменных)
```

переменных и область видимости

```
using System;
class Example {
    static void Main() {

        int x; //это переменная доступная для всего кода внутри метода Main()

        int y = 20; //аналогично предыдущей, но с инициализацией

        x = 10; //Отдельное присваивание числа переменной

        if (x == 10) { //начать новую область действия

            //int y = 40; //так делать нельзя,
            //т.к. уже есть родительская переменная с этим именем (отличие от C++)

            int z = 40; //Эта переменная доступна только внутри блока if

            //Здесь доступны обе переменные
            x = y * 2;

            //Динамическая инициализация переменной
            double xy = x / z;
        } //конец области действия блока if
        // z = 100; //так делать тоже нельзя
        //здесь переменная z уже за пределами области видимости

        //А переменная x здесь по-прежнему доступна
        Console.WriteLine("x = " + x);
    }
}
```

Формат записи при выводе

Метод WriteLine() для отображения строк, значений переменных и т.д. может использоваться в двух вариантах:

1) Между выводящимися частями ставится «+», и тогда получается нечто следующее:

Console.WriteLine(“Его возраст: ” + age + “ лет.”);

Его возраст 25 лет.

2) Между выводящимися частями ставится «,», и тогда всё меняется: первым аргументом передаётся форматирующая строка в двойных кавычках, а через запятую передаются остальные аргументы. Данный вывод чем-то похож на си-подобный подобный вывод printf, но использовать printf немного сложнее.

Пример форматированного вывода C# № 1:

Console.WriteLine(“В феврале {0} или {1} дней.”, 28, 29);

В феврале 28 или 29 дней.

Здесь аргументы вставляются вместо индексов в фигурной скобке.

Следующий пример задаёт ширину полей.

Пример форматированного вывода C# № 2:

Console.WriteLine(“В феврале {0,10} или {1,5} дней.”, 28, 29);

В феврале 28 или 29 дней.

Здесь вторая цифра в фигурных скобках как раз и задаёт формат полей, поэтому если число занимает 2 символа, то спереди будут добавлены ещё 8 символов в первом случае и 3 символа во втором случае примера № 2.

Формат записи при выводе

В следующем примере спецификатор формата использует управляющие последовательности, в данном случае горизонтальную табуляцию \t (Об управляющих символах будет подробнее рассказано позже).

Пример форматированного вывода C# № 3:

```
Console.WriteLine("Число \t Квадрат \t Куб");
for(i = 1; i < 10; i++)
    Console.WriteLine("{0} \t {1} \t {2}", i, i*i, i*i*i);
```

Результат выполнения выглядит следующим образом:

```
Число   Квадрат   Куб
1    1    1
2    4    8
3    9    27
4    16   64
5    25   125
6    36   216
7    49   343
8    64   512
9    81   729
```

Формат записи при выводе

Следующий пример устанавливает количество знаков после запятой.

Пример форматированного вывода C# № 4:

Console.WriteLine(“Деление 10/3 даёт: {0:#.##}”, 10.0 / 3.0);

Деление 10/3 даёт: 3.33

Ноль в фигурных скобках – это первый и единственный аргумент (результат деления), а через двоеточие методу указывается отобразить два десятичных разряда в дробной части числа.

Следующий пример задаёт разделители целой части, например, запятую.

Пример форматированного вывода C# № 5:

Console.WriteLine(“ {0:###,###.##}”, 123456.56);

123,456.56

Для вывода денежных сумм, например, рекомендуется использовать спецификатор формата C.

decimal balance;

balance = 12323.09M;

Console.WriteLine(“Текущий баланс равен {0:C}”, balance);

Результат выполнения этого фрагмента кода выводится в формате денежных сумм, указываемых в долларах США (для иностранных версий Windows) и рублях (например, для русскоязычных, если системой это предопределено).

Текущий баланс равен $12,323.09

Суффиксы для обозначения типов констант

Число 22022 может быть сохранено в типе int, но если добавить суффикс 22022L или 22022l, то число будет типа long.

Суффиксы:

22022u и 22022U - тип uint;

22022UL и 22022ul – тип ulong;

22022L и 22022l – тип long;

1.234f и 1.234F - тип float

1.234D и 1.234d - тип double (избыточно, т.к. по умолчанию)

1.234m и 1.234M - тип Decimal (обязательно для указывания)

Шестнадцатеричные литералы:

count = 0xFF; //255 в десятичной системе

incr = 0x1a; //26 в десятичной системе

Коды управляющих последовательностей

Управляющая последовательность	Описание
\a	Звуковой сигнал (звонок)
\b	Возврат на одну позицию
\f	Перевод страницы (переход на новую страницу)
\n	Новая строка (перевод строки)
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\o	Пустой символ
\’	Одинарная кавычка
\”	Двойная кавычка
\\	Обратная косая черта

Правила преобразования типов (по «рангу» операнда сверху вниз)

один из операндов	второй операнд	исключения
decimal	2 op -> decimal	~~float, double~~
double	2 op -> double	
float	2 op -> float	
ulong	2 op -> ulong	~~sbyte, short, int, long~~
long	2 op -> long	
uint	sbyte, short, int	1 op, 2 op -> long
uint	кроме sbyte, short, int: 2 op -> uint	
остальные операнды	остальные операнды	1 op, 2 op -> int

Первая строка интерпретируется следующим образом:
«ЕСЛИ один операнд имеет тип decimal, ТО и второй операнд продвигается к типу decimal, но если второй операнд имеет тип float или double, результат будет ошибочным»

Преобразование и приведение типов

```
using System;
class Example {
    static void Main() {
        int i = 10;
        double d = 12.64; //можно написать и 12.64D
        float f = 10.52; //так недопустимо, поскольку 10.52 - это тип Double
        float ff = 10.52F; //а так допустимо, у числа указан литерал float
        ff = (float) 10.52; //явное приведение, так тоже можно
        ff = (float)d; //и так тоже можно, но с потерей точности
        i = ff; //Недопустимое пробразование из float в int
        i = (int)ff; //явное приведение, но с потерей точности
        d = i; //А int в double (или float) можно (автоматическое преобразование)
        long L = 23564;
        d = L; //допустимо (автоматическое преобразование)
        L = d; //недопустимо
    }
}
```

Необходимое приведение типов

Из 25 слайда (последней строки) следует, что некоторые операнды будут приведены в int, даже если будут изначально иметь другой тип, например bool или char, а также все типы, «меньшие» int.

Пример № 1:

bool b, bb=true;

b = bb * bb; - это ошибка! Произойдёт преобразование в int.

b = (byte) (bb * bb); - а здесь ошибки нет. Необходимо явное приведение.

Пример № 2:

char ch1 = ‘a’, ch2 = ‘b’;

ch1 = (char) (ch1 + ch2); //Только так

Математические функции (класс System.Math)

Функция	Описание	Возвращаемое значение
Тригонометрические функции		
Math.Acos(A)	Арккосинус числа A	double
Math.Asin(A)	Арксинус числа A	double
Math.Atan(A)	Арктангенс числа A	double
Math.cos(A)	Косинус числа A	double
Math.sin(A)	Синус числа A	double
Math.tan(A)	Тангенс числа A	double
Функции округления и взятия остатка от деления		
Math.Ceiling(A)	«Округление вверх» числа A	**double**
Math.Floor(A)	«Округление вниз» числа A	**double**
Math.Round(A)	Округление по математическим правилам числа A (см. пример)	Присваиваемое любому подходящему численному типу
Math.Truncate(A)	«Обрезает» дробную часть числа A	**double**
Math.IEEERemainder(A, B)	Подобен оператору %, но для чисел с плавающей точкой (double)	**double**

Математические функции (класс System.Math)

Функция	Описание	Возвращаемое значение
Логарифмические функции		
Log(A, B)	Логарифм числа A по основанию B	double
Log10(A)	Логарифм числа A по основанию 10	double
Функции сравнения и нахождения максимального / минимального объекта		
Math.Equals(A,B)	Сравнивает объекты A и B	bool
Math.Max(A,B)	Возвращает большее из двух чисел	Присваиваемое любому подходящему численному типу
Math.Min(A,B)	Возвращает большее из двух чисел	Присваиваемое любому подходящему численному типу
Константы		
Math.E	Число E (основание натурального логарифма)	double
Math.PI	Число Пи	double
Другие функции		
Math.Abs(A)	Возвращает абсолютное представление A (модуль)	Присваиваемое любому подходящему численному типу
Math.Exp(A)	Возвращает константу E, возведенную в степень A	double
Math.Pow(A, B)	Возвращает A в степени B	double
Math.Sign(A);	Возвращает значение, определяющее знак числа A	int (-1 - минус, 0 - ноль, 1 - плюс)
Math.Sqrt(A)	Возвращает квадратный корень числа A	double

математических функции в C# (на примере)

```
using System;
class Example {
    static void Main() {
        int i = -100;
        double d = 12.5;

        int ii = Math.Abs(i);
        Console.WriteLine(ii); //100 (модуль числа)

        double dd = Math.Round(d);
        Console.WriteLine(dd); //12 (округление), но 12.51 дало бы 13

        d = 12.31;
        dd = Math.Ceiling(d);
        Console.WriteLine(dd); //13 (добавление до большего числа)

        d = 12.71;
        dd = Math.Floor(d);
        Console.WriteLine(dd); //12 (убавление до меньшего числа)

        dd = Math.Sqrt(ii);
        Console.WriteLine(dd); //10

        dd = Math.E;
        Console.WriteLine(dd); //основание натурального логарифма E

        dd = Math.PI;
        Console.WriteLine(dd); //Число Пи

        dd = Math.Pow(i,2);
        Console.WriteLine(dd); //степень числа

        Console.ReadLine();
    }
}
```

file:///D:/SYSTEM/Types

```
100
12
13
12
10
2,71828182845905
3,14159265358979
10000
```

if и оператор выбора switch

1) Условный оператор if

2) Распространённые условия проверки

3) Оператор выбора switch

Условный оператор if

Описание	Код

Условный оператор с одним действием (операторные скобки не требуются). Дословно: если **a** больше **b** (т.е. условие является true (истиной), то присвоить переменной **a** значение **b**. Табуляция применена для удобства чтения. Пробелы и переносы строк не влияют на компилятор.	`if (a>b) a=b;`
	`//или`
	`if (a>b)`
	`  a=b;`

Условный оператор с несколькими действиями (операторные скобки требуются). Дословно: если **a** больше **b**, то **a** присвоить **b**, а **b** присвоить **b*b**.	`if (a>b) {`
	`  a = b;`
	`  b = b * b;`
	`}`

Если условий несколько, то они перечисляются через **else if**. Условие «по умолчанию» (если ничто не подходит под условие) записывается как **else**. Дословно: если **a** больше **b**, то **a** присвоить **b**, ещё если **b** больше **a**, то присвоить **b** значение **a**, а если ни одно из условий не выполняется (т.е. например **a=b**), то **a** и **b** присвоить нулевые значения. Каждый оператор разделяется точкой с запятой. Данная конструкция более экономичная, т.к. после удачной первой (второй, …) проверки последующие проверки не осуществляются, что экономит время выполнения программы.	`if (a>b) a=b;`
	`else if (b>a) b=a;`
	`else {`
	`a = 0;`
	`b = 0;`
	`}`

проверки

Описание	Код
`if(a) {…}` то же, что `if(a != 0) {…}`	Условие выполняется, если переменная «а» не равна 0.
`if(a == 0) {…}`	Если «а» равно 0
`if(a = 0) {…}`	Бессмысленная, но синтаксически верная запись, компилятор не выдаст ошибку.
`if(a % 2 != 0) {…}`	Если «а» нечетное …
`if(a % 2 == 0) {…}`	Если «а» четное …

`x>0 ? y=x : x=y;`
то же самое, что:
`if(x>0) y=x;`
`else x=y;`
	Сокращенная форма if-else (тернарный оператор). Если «х» больше нуля, то присвоить «у» значение «х», в противном случае, присвоить «х» значение «у».

Оператор выбора switch

Описание	Код

Оператор основан на точном совпадении! Нельзя указывать диапазон значений, как, например, в Delphi или Pascal. После каждого совпадения необходим оператор **break**;, чтобы запретить программе выполнение остальных операторов case. В данном случае выполнится строка после **case 2:** (удвоение переменной a) Действия, выполняемые при несовпадении ни с одним условием, можно заключить в блоке **default:**	`int a = 2;`
	`switch (a) {`
	`  case 1:  a * a;`
	`  break;`
	`  case 2:  a = a * 2;`
	`  break;`
	`  case 5: // и т.д.`
	`  break;`
	`  default: exit;`
	`}`

Тот же оператор, но написанный без оператора **break;** будет выполнять все команды из блока switch-case, начиная с **case 2: (и вниз по списку)**	`int a = 2;`
	`switch (a) {`
	`  case 1:  a * a;`
	`  case 2:  a = a * 2;`
	`  case 5: // и т.д.`
	`  default: exit;`
	`}`

5. Циклы for, while, do while

1) Циклы
2) Цикл foreach
3) Оператор goto

Циклы

Название	for	while	do-while
Описание	for(инициализация; проверка; обновление) оператор;	while (условие true) оператор;	do оператор while(условие true);
Особенность	Условие проверяется до выполнения оператора. Объявлять и инициализировать переменную можно в разделе «инициализация», при этом «жизнь» переменной будет ограничиваться телом цикла.	Условие проверяется до выполнения оператора	Сначала выполняется действие, только по том проверяется условие. именно поэтому данный цикл выполняется, как минимум, 1 раз.

Пример с одним оператором (операторные скобки не обязательны)	`for(int i=1;i<=10;i++)`
	`Console.Write(i);`
	Вывод:
	12345678910
		`int i=1;`
		`while(i<=10)`
		`Console.Write(i);`
		Вывод:
		12345678910
			`int i=11;`
			`do Console.Write(i);`
			`while(i>=11);`
			Вывод:
			11 //т.к. условие проверяется в конце

Пример с несколькими операторами (операторные скобки обязательны)	`for(int i=1;i<=10;i++) {`
	`Console.Write(i);`
	`a += 5;`
	`b = a;`
	`}`
		`int i=1;`
		`while(i<=10) {`
		`Console.Write(i);`
		`a += 5;`
		`b = a;`
		`}`
			`int i=1;`
			`do {`
			`Console.Write(i);`
			`a += 5;`
			`b = a;`
			`} while(i<=10);`

Ещё особенности	`for( ; ; ) //бесконечный цикл`
	`for(int i=1, int j=2; i<=10, j>0; i++, j--) //тоже рабочий вариант`
		`while ( )`
		`Console.Write(i);`
		`//тоже бесконечный цикл`
			`do Console.Write(i) while ( );`
			`//тоже бесконечный цикл`

Оператор break; служит для преждевременного выхода из цикла	`for(int i=1;i<=10;i++) {`
	`if a == 5 Console.Write(i);`
	`else break;`
	`}`
		`int i=1;`
		`while(i<=10) {`
		`if a == 5`
		`Console.Write(i);`
		`else break;`
		`}`
			`int i=1;`
			`do {`
			`if a == 5`
			`Cosnsole.Write(i);`
			`else break;`
			`} while(i<=10);`

Наряду с оператором break;, оператор **continue;** заставляет пропустить все оставшиеся операторы и сразу же перейти к следующей итерации цикла.

Цикл foreach

Для удобства обработки «коллекций» (например, массивов) создан цикл foreach, основанный на диапазоне:

```
short [] seasons = new short[] {1,2,3,4}; //про массивы будет сказано позже
foreach (double x in seasons)
Console.Writeline(x);
```

Здесь цикл выведет каждый элемент массива один за другим.

Следует, однако, иметь в виду, что в отличие от C++, переменная цикла (в контексте это double x) в операторе foreach языка C# служит только для чтения. Это означает, что, присваивая этой переменной новое значение, нельзя изменить содержимое массива.

Оператор foreach можно использовать только для типов, наследующих интерфейс перечисления *IEnumerable<T>* (Данная тема выходит за рамки данной презентации)

Оператор goto

Нелюбимый программистами оператор безусловного перехода goto также имеется в C#. Данный оператор используется очень редко, т.к. способствует созданию т.н. «макаронного кода», т.к. благодаря ему в большой программе очень тяжело разобраться, что за чем выполняется, если данный оператор находится в неумелых руках.

Когда в программе встречается оператор goto, её выполнение переходит непосредственно к тому месту, на которое указывает этот оператор. Для этого в программе создаётся метка перехода:

```
using System;
class Example {
    static void Main() {
        int x = 1;
    loop1: //метка
        x++;
        if (x < 100) goto loop1;
        Console.WriteLine(x); //вывод 100
        Console.ReadLine();
    }
}
```

```
using System;
class Example {
    static void Main() {
        int x = 1;
        while (x < 100) x++; //Аналогично
        Console.WriteLine(x); //вывод 100
        Console.ReadLine();
    }
}
```

лме й выход оженных циклов или при использовании рекурсивных функций (хотя в большинстве случаев можно обойтись и без этого оператора).

6. Массивы и строки

Массив представляет собой совокупность переменных одного типа с общим для обращения к ним именем

Одномерные массивы

Общий формат (**Отличается от такового в C++**):

Тип[] имяМассива = new Тип[размерМассива];

Пример:

```
short[] seasons = new short[4]; // массив из 4 элементов типа short.
```

Или:

```
short[] seasons;
seasons = new short[4];
```

Или с инициализацией:

```
short[] seasons = new short[4] {1,2,3,4};
```

Или инициализация без указания размера:

```
short[] seasons = new short[] {1,2,3,4};
```

Или отдельная инициализация:

```
short[] seasons;
seasons = new short[] {1,2,3,4};
//не допускается
int hand[5];
hand[5] =  {32,21,88,13};
Console.Writeline(seasons[0]); //вывод на консоль 1 элемента
Console.Writeline(seasons[1]); //вывод на консоль 2 элемента
Console.Writeline(seasons); //без указания индекса в выводе будет указана информация об объекте, например, System.Int16[]
```

Границы массива в C# строго соблюдаются (***отличие от C++***). Если границы массивы не достигаются или же превышаются, то возникает ошибка при выполнении.

Многомерные массивы с фиксированной длиной

Объявление двумерного массива:

```
int [,] mas = new int [4,5];
```

mas [0,0]	mas [0,1]	mas [0,2]	mas [0,3]	mas [0,4]
mas [1,0]	mas [1,1]	mas [1,2]	mas [1,3]	mas [1,4]
mas [2,0]	mas [2,1]	mas [2,2]	mas [2,3]	mas [2,4]
mas [3,0]	mas [3,1]	mas [3,2]	mas [3,3]	mas [3,4]

Вывод массива:

```
for (int row = 0; row <4; row++) {
    for (int col = 0; col <5; ++col)
        Console.Writeline(mas [row,col] + “ “);
}
```

Инициализация массива:

```
int [,] mas =
{
    {96,100,87,101,105}, // значения для mas[0]
    {96,98,91,107,104},  // значения для mas[1]
    {97,101,91,108,107}, // значения для mas[2]
    {98,103,95,109,108}, // значения для mas[3]

}
```

Многомерные ступенчатые (зубчатые) массивы

Ступенчатый массив представляет собой массив массивов, в котором длина каждого массива может быть разной. Такой тип массива в C++ отсутствует.

Объявление двумерного ступенчатого массива:

```
int [][] mas = new int [3][];
mas[0] = new int[4];
mas[1] = new int[3];
mas[2] = new int[5];
```

После выполнения этого фрагмента кода массив jagged выглядит так, как показано ниже.

mas[0][0]	mas[0][1]	mas[0][2]	mas[0][3]	
mas[1][0]	mas[1][1]	mas[1][2]		
mas[2][0]	mas[2][1]	mas[2][2]	mas[2][3]	mas[2][4]

массивы

```
1  using System;
2  class AssignARef {
3      static void Main() {
4          int i;
5
6          int[] nums1 = new int[10];
7          int[] nums2 = new int[10];
8
9          for (i = 0; i < 10; i++) nums1[i] = i;
10         for (i = 0; i < 10; i++) nums2[i] = -i;
11         Console.Write("Содержимое массива nums1: ");
12         for (i = 0; i < 10; i++) Console.Write(nums1[i] + " ");
13         Console.WriteLine();
14         Console.Write("Содержимое массива nums2: ");
15         for (i = 0; i < 10; i++) Console.Write(nums2[i] + " ");
16         Console.WriteLine();
17
18         nums2 = nums1;
19
20         Console.Write("Содержимое массива nums2: ");
21         for (i = 0; i < 10; i++) Console.Write(nums2[i] + " ");
22         Console.WriteLine();
23
24         //Далее оперировать массивом nums1 посредством переменной ссылки
25         //на массив nums2.
26         nums2[3] = 99;
27
28         Console.Write("Содержимое массива nums1 после изменения\n" +
29                       "посредством переменной nums2: ");
30         for (i = 0; i < 10; i++) Console.Write(nums1[i] + " ");
31         Console.ReadLine();
32     }
33 }
```

```
Содержимое массива nums1: 0 1 2 3 4 5 6 7 8 9
Содержимое массива nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Содержимое массива nums2: 0 1 2 3 4 5 6 7 8 9
Содержимое массива nums1 после изменения
посредством переменной nums2: 0 1 2 99 4 5 6 7 8 9
```

Свойство Length

Свойство Length связано с каждым массивом и содержит число элементов, из которых может состоять массив. Ключевое слово - **<u>может</u>** - говорит о том, что на самом деле в массиве может быть задействовано меньшее количество элементов, но его «длина» всегда будет одинаковой.

```
1   using System;
2   class LengthDemo {
3       static void Main() {
4           int[] nums = new int[10];
5
6           Console.WriteLine("Длина массива nums равна " + nums.Length);
7
8           //Использовать свойство Length для инициализации массива nums
9           for (int i = 0; i < nums.Length; i++) nums[i] = i * i;
10
11          //А теперь воспользоваться свойством Length для вывода
12          //содержимого массива nums.
13          for (int i = 0; i < nums.Length; i++) Console.Write(nums[i] + " ");
14
15          Console.ReadLine();
16      }
17  }
```

file:///c:/users/бабин/documents/visual studio 2012/Pr

```
Длина массива nums равна 10
0 1 4 9 16 25 36 49 64 81
```

многомерных ступенчатых (зубчатых) массивов

В многомерном массиве свойство Length покажет общее кол-во элементов. В ступенчатом же массиве иначе: без указания индексации - покажет кол-во подмассивов, а с указанием индексации: размерность каждого подмассива.

```
1   using System;
2   class LengthDemo {
3       static void Main() {
4           //Двумерный массив
5           int[,] nums = new int[4,10];
6
7           //Покажет общее количество элементов (4*10 = 40)
8           Console.WriteLine("Длина массива nums равна " + nums.Length);
9
10          //Ступенчатый массив
11          int[][] nums2 = new int[4][];
12          nums2[0] = new int[3];
13          nums2[1] = new int[7];
14          nums2[2] = new int[2];
15          nums2[3] = new int[5];
16
17          //Покажет 4
18          Console.WriteLine("Общее количество подмассивов " + nums2.Length);
19
20          //А здесь покажет размерность каждого подмассива
21          Console.Write("Общее количество подмассивов " + nums2[0].Length);
22          Console.WriteLine();
23          Console.Write("Общее количество подмассивов " + nums2[1].Length);
24          Console.WriteLine();
25          Console.Write("Общее количество подмассивов " + nums2[2].Length);
26          Console.WriteLine();
27          Console.Write("Общее количество подмассивов " + nums2[3].Length);
28          Console.ReadLine();
29      }
30  }
```

```
file:///c:/users/6абин/documents/visual st
Длина массива nums равна 40
Общее количество подмассивов 4
Общее количество подмассивов 3
Общее количество подмассивов 7
Общее количество подмассивов 2
Общее количество подмассивов 5
```

Неявно типизированные массивы

Как и в случае с неявно типизированной переменной, в C# допускается создавать неявно типизированные массивы при помощи ключевого слова var. Но в таком случае, как и с переменной, массив сразу же необходимо инициализировать значениями. Отсутствуют в C++.

```
//Одномерный массив
var vals = new[] {1,2,3,4,5};
//Двумерный массив
var vals = new[,] { {1,2}, {3,4}, {5,6} };
//Неявно типизированный ступенчатый массив
var vals = new[,] {
   {1,2,3,4},
   {9,8,7},
   {11,12,13,14,15}
 };
```

Строки

Строки в C# объявляются и инициализируются следующим образом.

1 вариант (прямая инициализация):

```
string str = “Строка в C#”;
```

2 вариант (инициализация через массив символов):

```
char[] charray = {‘t’, ‘e’, ‘s’, ‘t’};
string str = new string(charray);
```

3 вариант (инициализация пользователем):

```
string str;
str = Console.Read();
```

Вывод строки осуществляется так же, как и обычной переменной:

```
Console.Write(str);
```

```
1   using System;
2   class UsingStrings {
3       static void Main() {
4           string str1 = "Моя строка";
5
6           char[] charray = {'t', 'e', 's', 't'};
7           string str2 = new string(charray);
8
9           Console.Write("Введите строку: ");
10          string str3 = Console.ReadLine();
11
12          Console.WriteLine(str1);
13          Console.WriteLine(str2);
14          Console.WriteLine(str3);
15
16          Console.ReadLine();
17      }
18  }
```

file:///c:/users/бабин/docum

```
Введите строку: Привет
Моя строка
test
Привет
```

Операции со строками

Метод	Описание
int Compare(string *strA*, string *strB*, StringComparison *comparisonType*)	**Возвращает** отрицательное значение, если строка *strA* меньше строки *strB*; положительное значение, если строка *strA* больше строки *strB*; и нуль, если сравниваемые строки равны. Способ сравнения определяется аргументом *comparisonType*.
bool Equals(string *value*, StringComparison *comparisonType*)	**Возвращает** логическое значение true, если вызывающая строка имеет такое же значение, как и у аргумента *value*. Способ сравнения определяется аргументом *comparisonType*.
int IndexOf(char *value*)	Осуществляет поиск в вызывающей строке первого вхождения символа, определяемого аргументом *value*. Применяется порядковый способ поиска. Возвращает индекс первого совпадения с искомым символом или **-1**, если он не обнаружен.
int IndexOf(string *value*, StringComparison *comparisonType*)	Осуществляет поиск в вызывающей строке первого вхождения подстроки, определяемой аргументом *value*. Возвращает индекс первого совпадения с искомой подстрокой или **-1**, если она не обнаружена. Способ сравнения определяется аргументом *comparisonType*.
int LastIndexOf(char *value*)	Осуществляет поиск в вызывающей строке последнего вхождения символа, определяемого аргументом *value*. Применяется порядковый способ поиска. Возвращает индекс первого совпадения с искомым символом или **-1**, если он не обнаружен.
int LastIndexOf(string *value*, StringComparison *comparisonType*)	Осуществляет поиск в вызывающей строке последнего вхождения подстроки, определяемой аргументом *value*. Возвращает индекс первого совпадения с искомой подстрокой или **-1**, если она не обнаружена. Способ сравнения определяется аргументом *comparisonType*.

Операции со строками

Метод	Описание
string ToLower(CultureInfo.CurrentCulture *culture*)	Возвращает вариант вызывающей строки в нижнем регистре. Способ преобразования определяется аргументом *culture (не обязателен)*.
string ToUpper(CultureInfo.CurrentCulture *culture*)	Возвращает вариант вызывающей строки в верхнем регистре. Способ преобразования определяется аргументом *culture (не обязателен)*.
string Substring(int *индекс_начала*, int *длина*)	Возвращает строку с позиции *индекс_начала* в количестве символов, равных *длина*.
str1 = str2 + str3;	Конкатенация (объединение) строк.
str1 == str2	Сравнение строк без учета культурной среды. Возврат true, если строки равны и false, если строки не равны.
str1 != str2	Сравнение строк без учета культурной среды. Возврат false, если строки равны и true, если строки не равны.

Некоторые методы принимают параметр типа **StringComparison**. Это перечислимый тип, определяющий различные значения, которые определяют порядок сравнения символьных строк.

Как правило и за рядом исключений, для сравнения символьных строк с учетом культурной среды (т.е. языковых и региональных стандартов) применяется способ **StringComparison.CurrentCulture**. Если же нужно сравнить строки только на основании их символов, то лучше воспользоваться способом **StringComparison.Ordinal**, а для сравнения строк без учета регистра - одним из двух способов: **StringComparison.CurrentCultureIgnoreCase** или **StringComparison.OrdinalIgnoreCase**.

Пример операций со строками

```
1  using System;
2  using System.Globalization;
3  class StrOps
4  {
5      static void Main()
6      {
7          string str1 = "Программировать в .NET лучше всего на C#.";
8          string str2 = "Программировать в .NET лучше всего на C#.";
9          string str3 = "Строки в C# весьма эффективны.";
10         string strUp, strLow;
11         int result, idx;
12
13         Console.WriteLine("str1: " + str1);
14         Console.WriteLine("Длина строки str1: " + str1.Length);
15
16         //Создать варианты строки str1, набранные прописными и
17         //строчными буквами
18         strLow = str1.ToLower(CultureInfo.CurrentCulture);
19         strUp = str1.ToUpper(CultureInfo.CurrentCulture);
20         Console.WriteLine("Вариант строки str1, " +
21                           "набранный строчными буквами:\n" + strLow);
22         Console.WriteLine("Вариант строки str1, " +
23                           "набранный прописными буквами:\n" + strUp);
24
25         //Вывод строки str1 посимвольно
26         Console.WriteLine("Вывод строки str1 посимвольно.");
27         for (int i = 0; i < str1.Length; i++) Console.Write(str1[i]);
28         Console.WriteLine();
29
30         //Сравнить строки способом порядкового сравнения
31         if (str1 == str2) Console.WriteLine("str1 == str2");
32         else Console.WriteLine("str1 != str2");
33         if (str1 == str3) Console.WriteLine("str1 == str3");
34         else Console.WriteLine("str1 != str3");
35
36         //Сравнить строки с учетом культурной среды
37         result = string.Compare(str3, str1, StringComparison.CurrentCulture);
38         if (result == 0) Console.WriteLine("Строки str1 и str3 не равны");
39         else if (result < 0) Console.WriteLine("Строка str1 меньше строки str3");
40         else Console.WriteLine("Строка str1 больше строки str3");
41         Console.WriteLine();
42
43         //Присвоить новую строку переменной str2
44         str2 = "Один Два Три Один";
45
46         //поиск подстроки
47         idx = str2.IndexOf("Один", StringComparison.Ordinal);
48         Console.WriteLine("Индекс первого вхождения подстроки <Один>: " + idx);
49         idx = str2.LastIndexOf("Один", StringComparison.Ordinal);
50         Console.WriteLine("Индекс последнего вхождения подстроки <Один>: " + idx);
51         Console.ReadLine();
52     }
53 }
```

```
str1: Программировать в .NET лучше всего на C#.
Длина строки str1: 41
Вариант строки str1, набранный строчными буквами:
программировать в .net лучше всего на c#.
Вариант строки str1, набранный прописными буквами:
ПРОГРАММИРОВАТЬ В .NET ЛУЧШЕ ВСЕГО НА C#.
Вывод строки str1 посимвольно.
Программировать в .NET лучше всего на C#.
str1 == str2
str1 != str3
Строка str1 больше строки str3

Индекс первого вхождения подстроки <Один>: 0
Индекс последнего вхождения подстроки <Один>: 13
```

Массивы строк (на примере)

```
1   using System;
2
3   class ConvertDigitsToWords {
4       static void Main() {
5           int num, nextdigit, numdigits;
6           int[] n = new int[20];
7           string[] digits = { "нуль", "один", "два",
8                               "три", "четыре", "пять",
9                               "шесть", "семь", "восемь",
10                              "девять"};
11          num = 1908;
12          Console.WriteLine("Число: " + num);
13          Console.Write("Число словами: ");
14          nextdigit = 0; numdigits = 0;
15
16          //Получить отедльные цифры и сохранить их в массиве n
17          //Эти цифры сохраняются в обратном порядке
18          do {
19              nextdigit = num % 10;
20              n[numdigits] = nextdigit;
21              numdigits++;
22              num = num / 10;
23          } while (num > 0);
24          numdigits--;
25
26          //Вывести полученные слова
27          for (; numdigits >= 0; numdigits--)
28              Console.Write(digits[n[numdigits]] + " ");
29
30          Console.ReadLine();
31      }
32  }
```

```
file:///c:/users/бабин/documents/visual studio 2012/P
Число: 1908
Число словами: один девять нуль восемь
```

Конвертирование строк в числа и чисел в строки (на примере)

```
using System.Globalization; //Подключение обязательно!

namespace ConvertStrings
{
    class Program
    {
        static void Main(string[] args)
        {
            //Строки чисел
            //обязательно ЗАПЯТАЯ, а не точка
            string s_double = "55,61";
            //Запятая или точка, если использовать CultureInfo.InvariantCulture
            string s_double2 = "100.78";
            string s_int = "1024";
            Console.WriteLine(s_double + " " + s_int);

            //Числовые переменные (конвертация) 1 вариант
            double dec = Convert.ToDouble(s_double);
            int num = Convert.ToInt32(s_int);

            //Числовые переменные (конвертация) 2 вариант
            double dec2 = double.Parse(s_double2, CultureInfo.InvariantCulture);
            int num2 = int.Parse(s_int);

            //Перевод из чисел в строки 1 вариант ToString()
            Console.WriteLine(dec.ToString() + " " + num.ToString());

            //Перевод из чисел в строки 2 вариант ToString()
            s_double = Convert.ToString(dec2);
            s_int = Convert.ToString(num2);
            Console.WriteLine(s_double + " " + s_int);

            Console.ReadLine();
        }
    }
}
```

file:///D:/C#/ConvertStrings/

```
55,61 1024
55,61 1024
100,78 1024
```

Запятая в данном примере обязательна, потому что по умолчанию в русскоязычных версиях Windows десятичным разделителем является запятая, а не точка. Использовать точку (или запятую) всё же можно, только указав вторым параметром при конвертации **CultureInfo.InvariantCulture**, предварительно подключив пространство имён **using System.Globalization**; этот параметр укажет функции на то, что не следует использовать текущие настройки региональных и языковых параметров при конвертации.

Операция «дословной» строки

Иногда удобно представлять строку дословно, особенно, когда строки содержат специальные символы. Для этого перед самой строкой (перед двойными кавычками) ставится символ «собачки» @.

Пример

```
//Такая строка требует экранировать (дублировать) специальные символы (в примере - обратной косой черты)
string s = “C:\\Users\\Doucments\\”;
//А такая строка не требует
string s = @”C:\Users\Documents\”;
```

На заметку:

В C# версий 6.0 появился оператор интерполяции строк ($), позволяющий подставлять переменные прямо в строку в фигурные скобки. При необходимости использовать в строке символ «фигурные скобки» его необходимо продублировать.

```
int years = 32;
string s = $”Мне {years} года”; //Выведет «Мне 32 года»
```

А при использовании символов ‘$’ и ‘@’ одновременно можно вывести строку, не прибегая, например, к специальному символу переноса строки ‘\n’. При этом символ ‘$’ обязательно должен стоять первым.

```
int x = 2;
string s = $@”Это охватывает {
x} строк”;
```

7. Функции

Иногда на протяжении всей программы приходится выполнять однотипные действия. При этом код этого действия может состоять как из одной строки, так и из нескольких. Во избежание дублирования кода и для повышения удобочитаемости кода используются функции.

1) Главная функция программы Main()
2) Общее представление о функциях
3) Передача параметров в функцию
4) Возврат значения функцией
5) Перегрузка функций
6) Перегрузка функций (пример)
7) Шаблонные функции
8) Что делать, если функция должна уметь изменять передаваемые ей параметры? Передача параметров по ссылке с модификатором ref
9) Что, если от функции требуется возврат более чем одного значения? Использование модификатора out
10) Что если количество аргументов, которые нужно передать функции, заранее не известно? Использование переменного числа аргументов. Модификатор params[]
11) Рекурсивные функции

Главная функция программы Main()

Как вы уже заметили, любая программа на языке C# имеет функцию Main(). С этой функции программа начинает работать. Рассмотрим её подробнее:

```
class Program
{
	static void Main (string[] args)
	{
		//Код программы
	}
}
```

Ключевое слово **static** указывает на то, что данная функция является статичной, т.е. она принадлежит собственно типу (или классу, в котором она находится), а не конкретному объекту. В данном случае она принадлежит классу Program и может быть вызвана только так: Program.Main(), без создания экземпляра класса (о классах речь пойдёт позже). Да и, в принципе, это невозможно.

Ключевое слово **void** указывает на то, что данная функция не возвращает никакого значения по завершению программы. Хотя это и возможно (а в C++ такое даже более правильно).

Функция принимает массив параметров **string[] args**, что позволяет из консоли при запуске программы передавать параметры для их дальнейшего использования. Эта тема рассмотрена не будет, более того, можно использовать вариант без параметров:

```
static void Main ()
```

Обратите внимание, что функция **Main()** должна быть написана именно с заглавной буквы, а не ~~**main()**~~, как это делается в некоторых других языках программирования.

Модификаторы доступа public / private и т.д. пока рассматриваться не будут, т.к. мы не выходим за рамки одного всего лишь класса и одного консольного выражения. По умолчанию, и класс Program, и функция Main() являются открытыми (public).

С пользовательскими функциями (методами) дело обстоит похожим образом, о чем будет рассказано далее.

Общее представление о функциях

Давайте представим, что внутри программы нам несколько раз необходимо выводить на консоль линию из звёздочек (*), для чего будет написан следующий код:

```
Console.WriteLine("*************************************************************");
```

Или так:

```
for(int i = 0; i < 50; i++)
{
        Console.Write("*");
}
Console.WriteLine();
```

И этот код необходимо будет вставлять всегда, когда нам это будет нужно. Вместо этого можно написать функцию и вызвать её по имени, чтобы она занимала всего одну строку в коде:

```
printline();
```

Для этого предварительно необходимо будет описать данную функцию следующим образом:

```
static void printline()
{
        for (int i = 0; i < 50; i++)
        {
                Console.Write('*');
        }
        Console.WriteLine();
}
```

Обратите внимание, что данная функция обязательно должна быть статической **static.** Также в данном случае функция не принимает никаких параметров printline(), а также не возвращает никакого значения, или же, как говорят, возвращает **void**.

Дальше – больше.

Передача параметров в функцию

Теперь намного удобнее вызывать эту функцию, всего лишь используя одну строку.

```
printline();
```

А теперь давайте представим, что мы захотели слегка приукрасить наше приложение и выводить каждый раз разные символы в виде линии. Например, не звёздочку, а символ нижнего подчёркивания (_). Можно, конечно, написать ещё одну такую же функцию и назвать её как-нибудь в духе printline_(), а ту функцию переименовать в printline_star(), но с таким же успехом можно было бы и не писать функцию вовсе. Гораздо проще сделать так, чтобы все наши излишества обрабатывала всего лишь одна функция printline, но при этом бы она принимала в качестве параметра символ, используя который, она должна бы была вывести линию. Это можно сделать, добавив возможность передавать функции параметры, как показано ниже:

printline('_'); или printline('*') и т.д.

Но для этого потребуется переписать функцию следующим образом:

```
static void printline(char ch)
{
        for (int i = 0; i < 50; i++)
        {
                Console.Write(ch);
        }
        Console.WriteLine();
}
```

Теперь наша функция выводит строку из любых переданных ей символов типа char в количестве 50 штук.

Дальше – больше.

Передача параметров в функцию

А что если нам каждый раз будет необходимо выводить линию разной длины, т.е. не фиксировано 50, а то 20, то 100, а то и 1000. Для этого потребуется добавить второй параметр в нашу функцию, который будет указывать целое число как количество необходимых нам для вывода символов:

```
static void printline(char ch, int q)
{
        for (int i = 0; i < q; i++)
        {
                Console.Write(ch);
        }
        Console.WriteLine();
}
```

Теперь мы каждый раз должны дополнительно передавать в функцию ещё один параметр – требуемое количество выводимых символов:

printline('+', 100); или printline('-', 20);

Тем самым мы упростили себе работу и увеличили функциональность нашей программы.

Но на данном этапе мы задействовали только передачу параметров функции. Функция внутри что-то выполняет, но мы так и не использовали возможность функции возвращать какое-либо значение. Для этого лучше будем использовать другую по функционалу версию функции.

Возврат значения функцией

Теперь, когда мы знаем, как передавать значения в функцию, давайте реализуем функцию, которая находит квадрат двух чисел. Данная функция показательная, она отсутствует в стандартном классе Math языка C#, хотя имеется, например, в языке Pascal.

Итак, нам необходимо получить результат в таком виде:

int num = sqr(10);

где в переменную num будет заноситься квадрат передаваемого значения (в данном случае квадрат 10, т.е. 100). Для этого необходимо, чтобы функция принимала параметром целое число и возвращала квадрат целого числа:

```
static int printline(int num)
{
        return num * num;
}
```

Вместо возвращаемого **void** (т.е. ничего не возвращаемого) мы используем тип **int**, значение которого вычисляется в функции и возвращается с использованием оператора **return**. Обратите внимание, что параметр с именем num совпадает с именем переменной num, что не обязательно. В любом случае, это не вызовет конфликта имён, т.к. за пределами функции данный параметр «не виден», хотя мы и могли дать другое имя параметру, а также, например, не сразу выводить ответ, а использовать ещё одну локальную переменную внутри функции, хотя в данном случае это будет избыточным:

```
static int printline(int n)
{
        int num = n;
        return n * n;
}
```

Когда функция небольшая, такое использование не приветствуется, но если код функции достаточно громоздкий, то создание внутренних локальных переменных (с любым именем) облегчит восприятие кода.

Дальше – больше.

Перегрузка функций

Возврат квадрата целого числа – это хорошо. Но как быть, если нам нужен будет квадрат не только типа int, но и типа double, decimal и т.д.

Для этого выше написанная функция не подойдёт, т.к. она не универсальная, т.е. не шаблонная (разговор о шаблонных функциях пойдёт чуть позже).

Напишем ещё одну функцию, поменяв принимаемый и возвращаемый параметры на double. Это называется перегрузкой функции, поэтому две функции с одинаковым именем sqr вполне могут совместно существовать. Однако для этого необходимо помнить правила перегрузки функций:

Перегружаемая функция должна либо иметь разный ТИП ПРИНИМАЕМЫХ параметров, либо разное КОЛИЧЕСТВО принимаемых параметров. Одного лишь изменения выходного параметра недостаточно!

Ниже приведён код трёх функций, одна из которых вызовет проблему:

```
//Параметр типа int
static int sqr(int n)
{
    return n * n;
}
//Параметр типа double
static double sqr(double n)
{
    return n * n;
}
```

А вот создание такой функции «рядом» с предыдущими вызовет проблему, т.к. только по возвращаемому параметру перегружать функции нельзя.

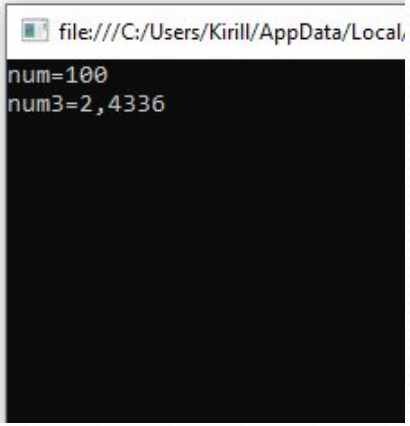
```
static double sqr(int n)
{
    return n * n;
}
```

И это легко объяснить: как компилятор определит, какая функция должна быть использована при передаче параметров одного типа? Никак.

Перегрузка функций (пример)

Ниже представлен пример программы, использующей перегрузку функций:

```
class Program
{

    static void Main(string[] args)
    {
        //Передача константы в функцию
        int num = sqr(10);
        double num2 = 1.56;
        //Передача переменной в функцию
        double num3 = sqr(num2);
        //Вывод результатов
        Console.WriteLine("num=" + num.ToString());
        Console.WriteLine("num3=" + num3.ToString());
        Console.ReadLine();
    }
    static int sqr(int n)
    {
        return n * n;
    }
    static double sqr(double n)
    {
        return n * n;
    }
}
```



Обратите внимание, что написанные нами функции вынесены из главной функции **Main()** и имеют ключевое слово **static**. Помещать функции внутри главного статического метода Main() нельзя. По крайней мере при рассмотрении данного примера в контексте версии языка C# 4.0

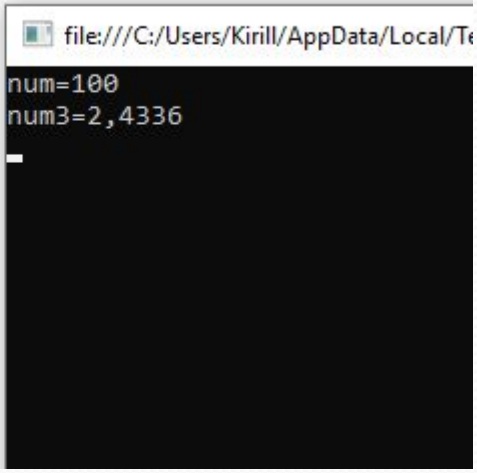
Шаблонные функции

Всё бы неплохо, но было бы гораздо лучше, чтобы существовала всего одна функция на все случаи жизни. Конечно, так бывает не всегда, но это возможно. И для этого существуют шаблонные функции. Справа представлен пример рабочей программы, в которой одна шаблонная функция возводит в квадрат числа разного типа.

Дело в том, что до запуска программы компилятору неизвестно, переменная какого типа будет передана в функцию. Поэтому условно тип указывается как «T» (имя «псевдотипа» может быть любым), а также внутри блока функции используется специальная переменная типа **dynamic**, т.к. во время выполнения программы происходит динамическое присваивание разных типов данных.

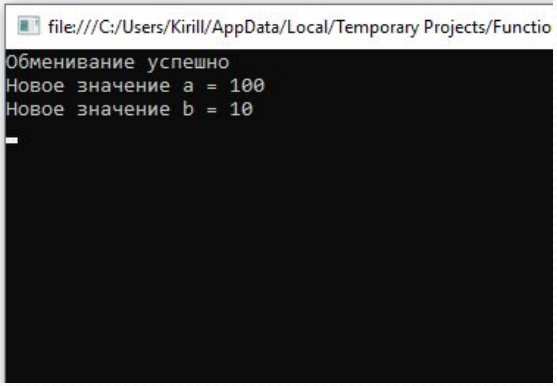
Примечание: Шаблонные функции работают медленнее обычных перегруженных функций, поэтому использование их оправдано только в действительно важных случаях.

```
static void Main(string[] args)
{
    //Передача константы в функцию
    int num = sqr(10);
    double num2 = 1.56;
    //Передача переменной в функцию
    double num3 = sqr(num2);
    //Вывод результатов
    Console.WriteLine("num=" + num.ToString());
    Console.WriteLine("num3=" + num3.ToString());
    Console.ReadLine();
}
static T sqr<T>(T num)
{
    dynamic n = num;
    dynamic result = n * n;
    return result;
}
```



Передача параметров по ссылке с использованием модификатора ref

```
static void Main(string[] args)
{
    int a = 10;
    int b = 100;
    bool result = successfull(ref a, ref b);
    Console.WriteLine(result ? "Обменивание успешно" : "Обменивание не произошло");
    if (result)
    {
        Console.WriteLine("Новое значение a = " + a.ToString());
        Console.WriteLine("Новое значение b = " + b.ToString());
    }
    Console.ReadLine();
}
static bool successfull(ref int n1, ref int n2)
{
    //Сделать первое число наибольшим, а второе наименьшим
    //Возвратить успешно произошло обменивание или нет
    if (n1 < n2)
    {
        int temp = n2;
        n2 = n1;
        n1 = temp;
        return true;
    }
    else
    {
        return false; //Обмен не произошёл
    }
}
```



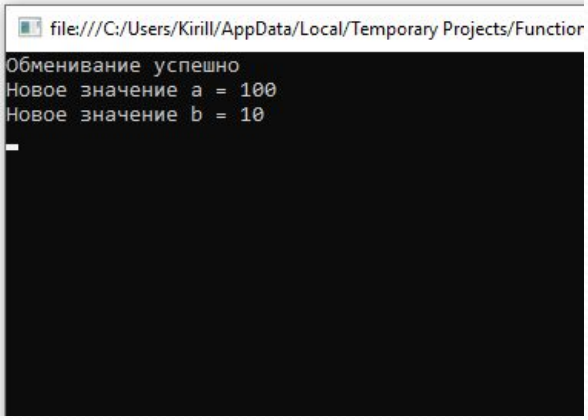
Иногда необходимо изменять значения передаваемых параметров в зависимости от условий. Слева приведен пример программы с функцией, в которую параметры передаются по ссылке на переменную в оперативной памяти, а не по значению (копии переменной). Поэтому функция не копирует значения, а может напрямую воздействовать их значения, переданных в виде параметра в качестве ссылки с использованием модификатора **ref**. Обратите внимание, что модификатор **ref** необходимо передавать как в описании функции, так и при её вызове.

Таким образом функция вернула истину, что обменивание успешно, а также изменила значения переменных.

При этом совершенно не обязательно изменять значения передаваемых параметров, чего нельзя сказать при использовании модификатора **out**, который будет рассматриваться далее.

Использование модификатора out

```
static void Main(string[] args)
{
    int a;
    int b;
    bool result = successfull(10, 100, out a, out b);
    Console.WriteLine(result ? "Обменивание успешно" : "Обменивание не произошло");
    if (result)
    {
        Console.WriteLine("Новое значение a = " + a.ToString());
        Console.WriteLine("Новое значение b = " + b.ToString());
    }
    Console.ReadLine();
}
static bool successfull(int a, int b, out int n1, out int n2)
{
    n1 = a;
    n2 = b;
    //Сделать первое число наибольшим, а второе наименьшим
    //Возвратить успешно произошло обменивание или нет
    if (n1 < n2)
    {
        int temp = n2;
        n2 = n1;
        n1 = temp;
        return true;
    }
    else
    {
        return false; //Обмен не произошёл
    }
}
```



В случае использования выходных параметров, на переменную также даётся ссылка в оперативной памяти, но с использованием модификатора **out**, при этом значение функцией игнорируется, и сравнить две переменные не получится.

Поэтому код был изменён так, чтобы перед проверкой сравниваемые значения были проинициализированы.

Параметры с модификатором **out** всегда должны стоять последними в списке передаваемых.

Конечно, данный пример слишком простой, т.к. использование параметров с модификатором **out** целесообразнее при возврате значений разных типов данных, в противном случае все значения можно было бы собрать в один массив, который бы и возвращала функция.

Использование переменного числа аргументов. Модификатор params[]

```
//Продемонстрировать применение модификатора params
//Функция находит минимальное значение среди переданных аргументов
using System;
class Min {
    public int MinVal(params int[] nums) {
        int m;
        if (nums.Length == 0) {
            Console.WriteLine("Ошибка: нет аргументов.");
            //0 возвращается при неправильном завершении программы
            return 0;
        }
        m = nums[0];
        for(int i=1; i < nums.Length; i++)
            if (nums[i] < m) m = nums[i];
        return m;
    }
}

class ParamsDemo {
    static void Main() {
        Min ob = new Min();
        int min;
        int a = 10, b = 20;

        //Вызвать метод с двумя значениями
        min = ob.MinVal(a,b);
        Console.WriteLine("Наименьшее значение равно " + min);

        //Вызвать метод с тремя значениями
        min = ob.MinVal(a, b, -1);
        Console.WriteLine("Наименьшее значение равно " + min);

        //Вызвать метод с пятью значениями
        min = ob.MinVal(18, 23, 3, 14, 25);
        Console.WriteLine("Наименьшее значение равно " + min);

        //Вызвать метод с массивом целых значений
        int[] args = {45, 67, 34, 9, 112, 8 };
        min = ob.MinVal(args);
        Console.WriteLine("Наименьшее значение равно " + min);
        Console.Read();
    }
}
```

```
file:///C:/Users/Администра...
Наименьшее значение равно 10
Наименьшее значение равно -1
Наименьшее значение равно 3
Наименьшее значение равно 8
```

Иногда в ходе выполнения программы количество аргументов может быть различно (1,2,3 и т.д.). Для такого случая используется модификатор params, указывается тип аргумента как массив с конкретным именем. Обращение же к каждому из передаваемых параметров осуществляется так же, как и к массиву, т.е. по индексу.

1) Необходимо обратить внимание, что все аргументы обязательно должны иметь тип массива, указанного в функции;
2) Вместе с переменным кол-вом параметров можно использовать обычные параметры, но массив с модификатором params всегда должен быть указан последним в списке параметров данного метода:

public void ShowArgs(string msg, params int[] nums) {...}

Рекурсивные функции

Рекурсивными функциями называются функции, вызывающими сами себя в процессе выполнения блока кода функции. Наглядным примером использования рекурсии может служить программа, находящая **факториал** числа. **Факториалом числа** n является произведение всех целых чисел от 1 до числа n включительно.

Записывается это так:

6! = 1 · 2 · 3 · 4 · 5 · 6 = 720

Справа представлена рекурсивная функция, находящая максимально возможный факториал числа (т.е. максимально возможное представление числа типа decimal). Такое число = 27.

```
using System;

namespace Factorial
{
    class Program
    {
        static decimal Factorial(decimal a)
        {
            if (a == 0) return 1;
            return a * Factorial(a - 1);
        }

        static void Main(string[] args)
        {
            decimal b = 27;
            //Максимальное число decimal
            Console.WriteLine(decimal.MaxValue);
            //Факториал 27!
            Console.WriteLine(Factorial(b));
            Console.ReadLine();
        }
    }
}
```

file:///D:/C#/Factorial/Factorial/bin/Debug/Factorial.EXE

```
79228162514264337593543950335
10888869450418352160768000000
```

8. Работа с папками и файлами (+ обработка исключений)

1) Обработка исключений. Блок try-catch-finally
2) Директории (папки)
   — Проверка на существование, получение списка подпапок и создание папок;
   — Переименование и удаление папок.
3) Файлы
   — Проверка на существование, переименование, удаление файла, а также получение списка файлов в папке (на примере);
   — Чтение из файла в консоль;
   — Запись в файл и удаление файла;
4) Решение олимпиадного задания «Недопалиндромы» с использованием чтения и записи в файл.

Обработка исключений. Блок try-catch-finally

```
try
{
    Directory.Delete(@"C:\MyDirectory");
    Console.WriteLine("Каталог успешно удалён");
}
//Необязательный "конкретизированный"
//исключением DirectoryNotFoundException блок catch,
//Если возможно отсутствие файла
catch (DirectoryNotFoundException ex)
{
    Console.WriteLine("Каталог не найден! " +
        ex.Message);
}
//Обязательный при отсутствии предыдущего блока catch
//Отлавливает любые другие исключения
//С возможностью получения информации об ошибке
catch (Exception ex)
{
    Console.WriteLine("Каталог не удалён! " +
        ex.Message);
}
//Обязательный при отсутствии предыдущих блоков catch
//Отлавливает любые другие исключения
//Без возможности получения информации об ошибке
//подчёркивается зелёным, т.к. это, при наличии предыдущего
//блока catch, – недостижимый участок кода
catch
{
    Console.WriteLine("Каталог не удалён!");
}
//Необязательный блок finally
//Просто завершает приложение
finally
{
    Console.ReadLine();
    Environment.Exit(0);
}
```

```
file:///C:/Users/Kirill/AppData/Local/Temporary Projects/try_catch_finally/bin/Debug/try_cat
Каталог не найден! Не удалось найти часть пути "C:\MyDirectory".
```

Очень часто приходится выполнять такие действия в коде программы, которые могут выполниться, а могут и завершить программу с ошибкой.

Чтобы ошибка не вызвала аварийное завершение программы, используется конструкция **try-catch**, которая дополнительно может быть оснащена блоком **finally**.

При этом, если имеется блок **try**, то обязательно должен существовать хотя бы один блок **catch** (или несколько блоков **catch**, если необходимо отлавливать исключения (ошибки) разных типов).

Блок **finally** необходимо включать в код программы в том случае, если, независимо от того, было ли перехвачено исключение в блоке **try**, требуется выполнение определенных действий по завершении кода из блока **try**.

Примеры использования блоков **try-catch-finally**:

— При создании / удаления / переименования директории (папки);

— Чтение / удаление / запись в файл;

— Получение информации из сети Интернет и т.д.

Слева приведён код программы, использующий простую конструкцию **try-catch** при удалении файла. Подробнее речь о работе с файлами пойдёт далее.

Блок try предпринимает попытку удаления каталога MyDirectory на диске C, но каталог преднамеренно не создан, поэтому перехватывается первое исключение. Если не нужно конкретизировать причину «неудаления» файла, можно использовать конструкцию **catch(Exception ex)** с возможностью просмотра информации об ошибке или конструкцию **catch** без параметров и, соответственно, без возможности просмотра ошибки. В приведённом коде последняя конструкция **catch** является недостижимой, т.к. предыдущая конструкция и так перехватывает все исключения.

Конструкции catch необходимо располагать от более специфичных в отношении ошибок (перехватывающим конкретные исключения) к менее специфичным. Но, как минимум, **ОДНА** конструкция **catch** обязательно должна присутствовать.

Директории (папки)

```
//Подключение пространства имён
using System.IO;
…
//Путь к подпапке в папке с программой
if(Directory.Exists(“MyDirectory”) {…}
else {…}

//Путь к подпапке, полученный программно
if(Directory.Exists(Environment.CurrentDirectory) {…}

//Или полный путь
if(Directory.Exists(“C:\\MyFolder\\MyFolder2”) {…}

//Или полный путь с указанием буквальной строки
if(Directory.Exists(@“C:\MyFolder\MyFolder2”) {…}

//Получение списка папок в папке
foreach(string s in Directory.GetDirectories(“Путь”))
{
      Console.WriteLine(s);
}

//Создание подпапки в папке с программой
Directory.CreateDirectory(Environment.CurrentDirectory
+ @“\MyDirectory”);

//Создание подпапки в произвольном каталоге
//Будет создана подпапка MyFolder2, если путь
//”C:\MyFolder” существует
if(Directory.Exists(@“C:\MyFolder”)
{
 Directory.CreateDirectory(@“C:\MyFolder\MyFolder2”);
}
```

Для работы с папками необходимо подключить пространство имён input-output (IO):

using System.IO;

Статический класс Directory имеет множество методов для работы с каталогами.

— Для проверки существования папки используется метод Exists(), который ***принимает*** параметром либо **полный путь к папке**, либо **относительный** (когда папка расположена в директории с исполняемым файлом программы .exe) и ***возвращает*** **true**, если папка существует, и **false**, если папка отсутствует, тем самым позволяя использовать метод внутри блока if для проверки.

— Для получения списка подпапок в определённой папке используется метод GetDitrctories(), принимающий параметром также путь к целевой папке, и возвращающий массив типа string [], позволяя получить все ссылки на подпапки в цикле foreach.

— Для создания папки используется метод CreateDirectory(), принимающий параметром путь к папке, в которой нужно создать подпапку. Однако создание подпапки может вызвать ошибку, и программа завершится аварийно, если системой или пользователем указан атрибут «Только чтение», именно поэтому в реальных программах необходимо использовать блоки try-catch, внутрь которых и помещать код с созданием папки.

Проверка на существование папок, получение списка подпапок и создание папок

```
using System;
using System.IO; //Обязательно!

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            //Проверка наличия каталога MyDirectory, располагающегося в каталоге с программой
            if (Directory.Exists("MyDirectory"))
            {
                //Здесь этот каталог умышленно первый - GetValue(0)
                Console.WriteLine("Каталог " +
                    Directory.GetDirectories(Environment.CurrentDirectory).GetValue(0).ToString());
            }
            else
            {
                //Создание каталога MyDirectory при отсутствии
                Directory.CreateDirectory(Environment.CurrentDirectory + @"\MyDirectory");
                Console.WriteLine("Каталог отсутствует");
            }
            Console.WriteLine("____________________________________");

            //Создание другого каталога, если отсутствует
            Directory.CreateDirectory(Environment.CurrentDirectory + @"\MyDirectory2");
            //Проверка наличия нескольких каталогов, располагающихся в каталоге с программой
            foreach (string s in Directory.GetDirectories(Environment.CurrentDirectory))
            {
                Console.WriteLine(s);
            }
            Console.ReadLine();
        }
    }
}
```

file:///D:/CSharp/Folders/Folders/Folders/bin/Debug/Folders.EXE

```
Каталог D:\CSharp\Folders\Folders\Folders\bin\Debug\MyDirectory
____________________________________
D:\CSharp\Folders\Folders\Folders\bin\Debug\MyDirectory
D:\CSharp\Folders\Folders\Folders\bin\Debug\MyDirectory2
```

Слева приведён пример программы, в которой проверяется наличие каталога. Если каталог отсутствует, то он создаётся.

Конструкции try-catch-finally не используются, т.к. работают намного медленнее, чем проверка условным оператором if. Хотя создание каталога в реальном коде должно быть в блоке Try, чтобы программа не закончилась аварийно в случае неуспешного создания папки.

Переименование и удаление папок

```
using System;
using System.IO; //Обязательно!

namespace Rename_Delete_Directory
{
    class Program
    {
        static void Main(string[] args)
        {
            //Переименование папки
            if (Directory.Exists(@"C:\MyDirectory"))
            {
                Directory.Move(@"C:\MyDirectory", @"C:\Моя папка");
                Console.WriteLine("Папка переименована.");
            }
            else
            {
                Console.WriteLine("Папка не переименована.");
            }

            //Удаление папки
            if (Directory.Exists(@"C:\Моя папка"))
            {
                Directory.Delete(@"C:\Моя папка");
                Console.WriteLine("Папка удалена.");
            }
            else
            {
                Console.WriteLine("Папка не удалена.");
            }
            Console.ReadLine();
        }
    }
}
```

```
file:///D:/CSharp/Rename_Delete_Director
Папка переименована.
Папка удалена.
```

Для переименования папок используется метод Move(), а для удаления — метод Delete().

В приведённом слева коде существующая папка «MyDirectory» сначала переименовывается в «Моя папка», а затем удаляется.

Первым параметром в методе Move() указывается путь к файлу, который нужно переименовать, а вторым — путь с именем, которое необходимо применить к папке. При этом не стоит беспокоиться о внутри лежащих файлах, т.к. с ними (в конечном счёте) предприниматься какие-либо действия не будут.

Метод Delete() принимает всего лишь один параметр – это путь к файлу.

ВНИМАНИЕ: папка не будет удалена, если внутри имеются подпапки или файлы! Для удаления подпапок или фалов внутри папки необходимо использовать другой алгоритм, который удаляет сначала все файлы в подпапках и в самой папке, а потом уже пустые подпапки и саму папку.

Файлы

```
//Подключение пространства имён
using System.IO;
…
//Путь к подпапке в папке с программой
if(File.Exists("MyFile.txt") {…}
else {…}

//Путь к подпапке, полученный программно
if(File.Exists(Environment.CurrentDirectory) {…}

//Или полный путь
if(File.Exists("C:\\MyFolder\\MyFile.txt") {…}

//Или полный путь с указанием буквальной строки
if(File.Exists(@"C:\MyFolder\MyFile.txt") {…}

//Получение списка файлов в папке
foreach(string s in Directory.GetFiles("Путь"))
{
      Console.WriteLine(s);
}

//Создание файла в папке с программой
File.Create(@"MyFile.txt");

//Создание файла в произвольном каталоге
//Будет создан файл MyFile.txt
File.Create (@"C:\MyFolder\MyFile.txt");
```

Для работы с файлами необходимо также подключить пространство имён input-output (IO):

using System.IO;

Статический класс File также, как и класс Directory, имеет множество методов для работы с файлами.

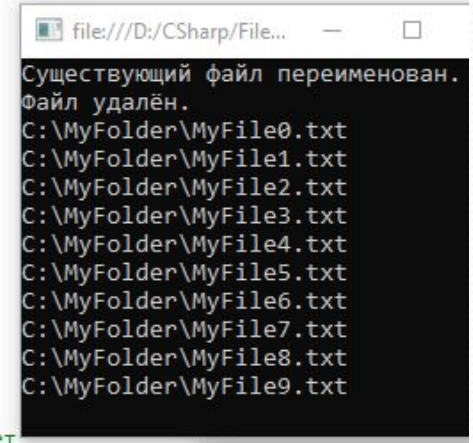
— Для проверки существования файла используется метод Exists(), который *принимает* параметром либо **полный путь к файлу (с указанием РАСШИРЕНИЯ файла)**, либо **относительный** (когда файл расположен в директории с исполняемым файлом программы .exe) и *возвращает* **true**, если файл существует, и **false**, если файл отсутствует, тем самым позволяя использовать метод внутри блока if для проверки.

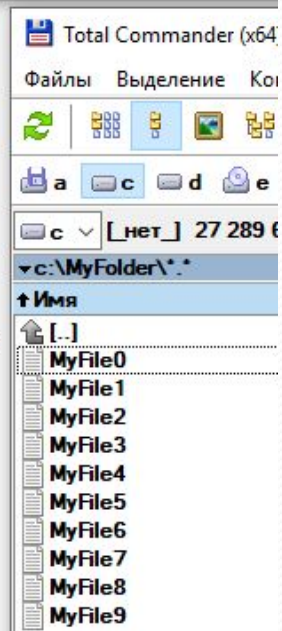
— Для получения списка файлов в определённой папке используется метод GetFiles(), принимающий параметром также путь к целевой папке, и возвращающий массив типа string [], позволяя получить все ссылки на файлы в цикле foreach.

— Для создания файла используется метод Create(), принимающий параметром путь к папке с указанием файла и расширения.

файла, а также получение списка файлов в папке (на примере)

```
//Проверка существования файла
//Если файл существует
if (File.Exists(@"C:\MyFolder\MyFile.txt"))
{
    //Переименовать файл
    File.Move(@"C:\MyFolder\MyFile.txt", @"C:\MyFolder\MyFile2.txt");
    Console.WriteLine("Существующий файл переименован.");
}
//Иначе создать файл
else
{
    File.Create(@"C:\MyFolder\MyFile2.txt");
    Console.WriteLine("Файл создан.");
}

//Если файл существует
if (File.Exists(@"C:\MyFolder\MyFile2.txt"))
{
    //Удалить файл
    File.Delete(@"C:\MyFolder\MyFile.txt");
    Console.WriteLine("Файл удалён.");
}
//Иначе вывести сообщение, что файл не существует
else
{
    Console.WriteLine("Файл не существует.");
}

//Создаёт файл в папке с программой
File.Create("MyFile.txt");

//Создать 10 файлов по указанному пути
for(int i = 0; i < 10; i++)
{
    File.Create(@"C:\MyFolder\MyFile" + i.ToString() + ".txt");
}

//Получение списка файлов в папке
foreach(string s in Directory.GetFiles(@"C:\MyFolder"))
{
    Console.WriteLine(s);
}
Console.ReadLine();
```





Чтение из файла в консоль

```
using System;
using System.Linq; //Для операции Select
using System.IO; //Для input-output чтения и записи в потоке

namespace FirstApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int A;
            double[] arr2;
            //Считывание всех числовых данных,
            //которые расположены каждый с новой строки
            //Если данные расположены через пробел или другой
            //символ(например, ~) - заменить '\n' на ' ' или '~' соответственно
            double[] arr =
                File.ReadAllText("input.txt").Split('\n').Select(n => double.Parse(n)).ToArray();
            //Считывание построчно, если в первой строке указано количество последующих строк
            //Поток закрывать не требуется, т.к. все объекты из блока using
            //впоследствии автоматически разрушаются
            using(StreamReader str = new StreamReader("input.txt"))
            {
                //Считывание первой строки, содержащей число количества последующих строк
                A = Convert.ToInt32(str.ReadLine());
                //Создание массива для вещественных чисел с количеством элементов A
                arr2 = new double[A];
                //Заполнение массива вещественными числами из последующих строк
                for (int i = 0; i < A; i++)
                    arr2[i] = Convert.ToDouble(str.ReadLine());
            }
            //Посмотреть содержимое 1 массива
            foreach (double d in arr) Console.WriteLine(d);
            //Разделитель (пустая строка)
            Console.WriteLine();
            //Посмотреть содержимое 2 массива
            foreach (double d in arr2) Console.WriteLine(d);
            Console.ReadLine();
        }
    }
}
```

file:///D:/CSharp/FirstApp/FirstApp/Firs

```
5
13
43
24
15
100

13
43
24
15
100
```

В данном случае для чтения данных из файла используется LINQ (Язык интегрированных запросов). Поэтому, помимо добавления пространства имён **System.IO**, необходимо подключить **System.Linq**;

Статический класс **File** вызывает метод **ReadAllText()**, которому в параметрах передаётся путь к файлу (в данном случае в папке с программой), и который возвращает строку, выглядящую в случае расположения данных с новой строки примерно в таком виде:

*5**\n**13**\n**43**\n**24**\n**15**\n**100*

Отсюда видно, что разделителем каждого числа (строки) является символ переноса на новую строку (**\n**), поэтому следующий метод Split(), который имеет любая строка типа string, разделяет строку на массив строк string[]. Для этого в качестве разделителя в параметре метода указан символ переноса на новую строку: Split('\n'). Если бы данные необходимо было разделить, когда они отделены пробелом (или любым другим символом), то вместо символа переноса строки нужно бы было указать этот символ.

Следующий метод **Select()** является примером Linq-запросов. Данный метод сначала (в виде лямбда-выражения) получает некий объект n, а возвращает (=>) этот объект, приведённым к типу double методом **double.Parse()**. В конечном итоге из каждого этого объекта создаётся массив типа double с использованием метода **ToArray()**. В результате конечный массив double [] arr получает все значения текстового документа в иде значений double.

Продолжение – на следующем слайде.

Чтение из файла в консоль

```
using System;
using System.Linq; //Для операции Select
using System.IO; //Для input-output чтения и записи в потоке

namespace FirstApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int A;
            double[] arr2;
            //Считывание всех числовых данных,
            //которые расположены каждый с новой строки
            //Если данные расположены через пробел или другой
            //символ(например, ~) - заменить '\n' на ' ' или '~' соответственно
            double[] arr =
                File.ReadAllText("input.txt").Split('\n').Select(n => double.Parse(n)).ToArray();
            //Считывание построчно, если в первой строке указано количество последующих строк
            //Поток закрывать не требуется, т.к. все объекты из блока using
            //впоследствии автоматически разрушаются
            using(StreamReader str = new StreamReader("input.txt"))
            {
                //Считывание первой строки, содержащей число количества последующих строк
                A = Convert.ToInt32(str.ReadLine());
                //Создание массива для вещественных чисел с количеством элементов A
                arr2 = new double[A];
                //Заполнение массива вещественными числами из последующих строк
                for (int i = 0; i < A; i++)
                    arr2[i] = Convert.ToDouble(str.ReadLine());
            }
            //Посмотреть содержимое 1 массива
            foreach (double d in arr) Console.WriteLine(d);
            //Разделитель (пустая строка)
            Console.WriteLine();
            //Посмотреть содержимое 2 массива
            foreach (double d in arr2) Console.WriteLine(d);
            Console.ReadLine();
        }
    }
}
```

file:///D:/CSharp/FirstApp/FirstApp/Firs

```
5
13
43
24
15
100

13
43
24
15
100
```

Если в первой строке текстового файла (не обязательно с расширением *.txt) указано количество последующих чисел, то сначала необходимо прочитать эту строку, а потом с использованием её значения установить длину массива для последующего считывания данных.

Во втором случае используется оператор **using**, параметром которому передаётся выражение:

StreamReader str = new StreamReader(“input.txt”)

Таким образом, весь дальнейший код в блоке **using** будет выполняться, используя созданную потоковую «переменную» **str**, «привязанную» к файлу с именем «**input.txt**». При этом нет необходимости закрывать поток, как в *Pascal*, типа ~~Close(str)~~, потому что объект автоматически разрушается после завершения блока кода.

Что происходит в блоке кода:

— считывается первая строка и переводится значение в тип **int**;

— устанавливается длина ранее созданного экземпляра массива **arr2**;

— с использованием цикла **for** заполняется массив.

Дальнейшие действия с использованием оператора **foreach** выполнены для вывода на консоль.

Обратите внимание, что метод **ReadLine()** используется не только с консолью, а вообще для получения данных из любого потока. В нашем случае из «потока» **str**.

Запись в файл и удаление файла

```
using System;
using System.IO; //Для input-output чтения и записи в потоке

namespace FirstApp
{
    class Program
    {
        static void Main(string[] args)
        {
            //Создать массив и инициализировать его данными
            double[] ar = {12.54, 14.42, 0, 100};
            //Создать строку и инициализировать её
            string str = "Это моя строка";

            //Запись одиночной строки в файл
            File.WriteAllText("output.txt", str);

            //Запись из массива в файл
            using (StreamWriter sw = new StreamWriter("output.txt"))
            {
                //С новой строки каждое значение
                foreach (double d in ar)
                {
                    sw.WriteLine(d);
                }
                //Через пробел каждое значение
                foreach (double d in ar)
                {
                    sw.Write(d.ToString() + ' ');
                }
            }
            Console.ReadLine();
        }
    }
}
```

output — Блокнот

Файл Правка Формат Вид Справка

```
12,54
14,42
0
100
12,54 14,42 0 100
```

Для записи в файл необходимо подключить пространство имён **System.IO**;

Для записи отдельной строки можно использовать метод **WriteAllText()** статического класса **File**.

Для записи всего массива класс **File** не подходит, т.к. метод **WriteAllText()** выполняет разовую запись в файл, после чего файл закрывается. Поэтому, если бы использовался цикл ~~**foreach**~~, то в конечном файле осталось бы только последняя строка со значением 100.

Для записи массива данных лучше, как и в случае с чтением файлов (**StreamReader**), использовать потоковый класс **StreamWriter**, который можно создать прямо внутри параметров **using**, по окончанию блока кода которого объект **sw** будет автоматически разрушен (освобождён из памяти, а поток закрыт).

В первом случае используется метод **WriteLine()**, который записывает каждое значение массива с новой строки.

Во втором случае используется метод **Write()**, который пишет в файл одной строкой. Искусственно для записи каждого значения через пробел мы привели значение типа **double** к типу **string** (**d.ToString()**) и дополнительно добавили после каждого числа пробел (+ ‘ ‘). Вместо пробела можно использовать любой символ.

Для удаления файла можно использовать статический класс **File** и его метод **Delete()**, принимающий в качестве параметра путь к файлу в виде строки. Данное действие аналогично действию по удалению папки (каталога), поэтому более подробно здесь не рассматривается.

«Недопалиндромы» с использованием чтения и записи в файл (условие задания)

Задача А. Недопалиндромы **20 баллов**

Ограничение по времени, сек.	2
Ограничение по памяти, мегабайт	1
Имя входного файла	`palind.in`
Имя выходного файла	`palind.out`

Страшный зверь первобытной сказки Штуша-Кутуша очень не любил числа. Они вызывали у него подозрение. Однако все животные первобытной сказки что-нибудь да знали о них. Поэтому Штуша-Кутуша решил придумать такие числа, о которых животные ничего не знали. Он назвал их недопалиндромами. Числом недопалиндромом он считает такое число, сумма цифр которого является палиндромом, причём само исходное число таковым не является. Например, таким недопалиндромом является число 137, так как его сумма цифр равна 11. Число 11 – палиндром, так как читается одинаково слева-направо и наоборот. Число 22 уже не будет недопалиндромом (несмотря на то, что сумма цифр – 4 – палиндром), так как само является палиндромом. Помогите Штуше-Кутуше найти такие числа.

Требуется написать программу, выводящую все недопалиндромы в указанном интервале.

Формат входного файла

В первой строке – два натуральных числа a, b ($1 \le a, b \le 32767$) – интервал чисел для поиска недопалиндромов.

Формат выходного файла

Выходной файл должен содержать N строк – числа недопалиндромы в интервале [a, b] (включая a и b), либо сообщение «NO», если таковых нет.

Пример

palind.in	**palind.out**
820 840	821 830

«Недопалиндромы» с использованием чтения и записи в файл

```
using System;
using System.Linq; //Для Select
using System.IO; //для ввода-вывода
using System.Collections.Generic; //Для List
using System.Diagnostics; //Для счёта времени выполнения кода

namespace Unterpalindroms
{
    class Program
    {
        //Функция, проверяющая, является ли число палиндромом
        public static bool isPalindrome(string s)
        {
            return s == new string(s.Reverse().ToArray());
        }
        //Функция, находящая сумму цифр в числовой строке
        public static int summa(string s)
        {
            int sum = 0;
            for (int i = 0; i < s.Length; i++)
                sum += Convert.ToInt32(s[i].ToString());
            return sum;
        }
        static void Main(string[] args)
        {
            //Измерить время выполнения кода
            Stopwatch SW = new Stopwatch(); // Создаем объект
            SW.Start(); // Запускаем

            //Читаем файл и получаем массив из двух чисел
            //Первое число - начальное (индекс 0), второе - конечное (индекс 1)
            int[] interval =
                File.ReadAllText("palind.in").Split(' ').Select(n => int.Parse(n)).ToArray();
            //Считаем количество чисел в этом интервале
            int count = interval[1] - interval[0] + 1;
            //Создаём массив, размером count
            int[] numbers = new int[count];
            //Заполняем массив всеми числами из интервала
            //Сразу же используется несколько переменных
            for (int i = 0, num = interval[0]; i < count; i++, num++)
                numbers[i] = num;
```

```
            //Создаём список для хранения всех недопалиндромов
            //Список саморасширяющийся, поэтому лучше использовать его
            //Т.к. заранее мы не знаем количество недопалиндромов
            List<int> upal = new List<int>();
            //Заполняем список
            foreach (int n in numbers)
            {
                if (!isPalindrome(n.ToString()) && isPalindrome(summa(n.ToString()).ToString()))
                    upal.Add(n);
            }
            //Записываем в файл
            using (StreamWriter sw = new StreamWriter("palind.out"))
            {
                //Если список не пустой
                if (upal.Count > 0)
                {
                    foreach (int i in upal)
                        sw.WriteLine(i);
                }
                //Если список пустой
                else
                    sw.WriteLine("NO");
            }
            SW.Stop(); //Останавливаем
            Console.WriteLine("Затраченное время в мс: " + SW.ElapsedMilliseconds);
            Console.ReadLine();
        }
    }
}
```

d:\CSharp\Unterpalindroms\Unterpalindroms\bin\Debug*.*

Имя	Тип	Размер	Дата	Атриб
[..]		<Папка>	30.03.2019 14:26	—
palind	in	7	30.03.2019 14:47	-a--
palind	out	30 124	30.03.2019 14:52	-a--
Unterpalindroms	exe	6 656	30.03.2019 14:52	-a--
Unterpalindroms	pdb	15 872	30.03.2019 14:52	-a--
Unterpalindroms.vshost	exe	11 600	30.03.2019 14:53	-a--

file:///D:/CSharp/Unterpalindroms/Unterpalindroms/bin/Debug/Unterpalindroms.EXE

```
Затраченное время в мс: 86
```

Использован максимальный диапазон (1-32767)
Двухъядерный процессор (не новый).