Синтаксис объектно-ориентированного языка программирования C#

Преподаватель: Тазиева Рамиля Фаридовна

ЯЗЫК ПРОГРАММИРОВАНИЯ C#

C# — объектно-ориентированный язык программирования от компании Microsoft для работы на платформе .NET (Windows Forms, WPF, ASP.NET, Xamarin, Unity).

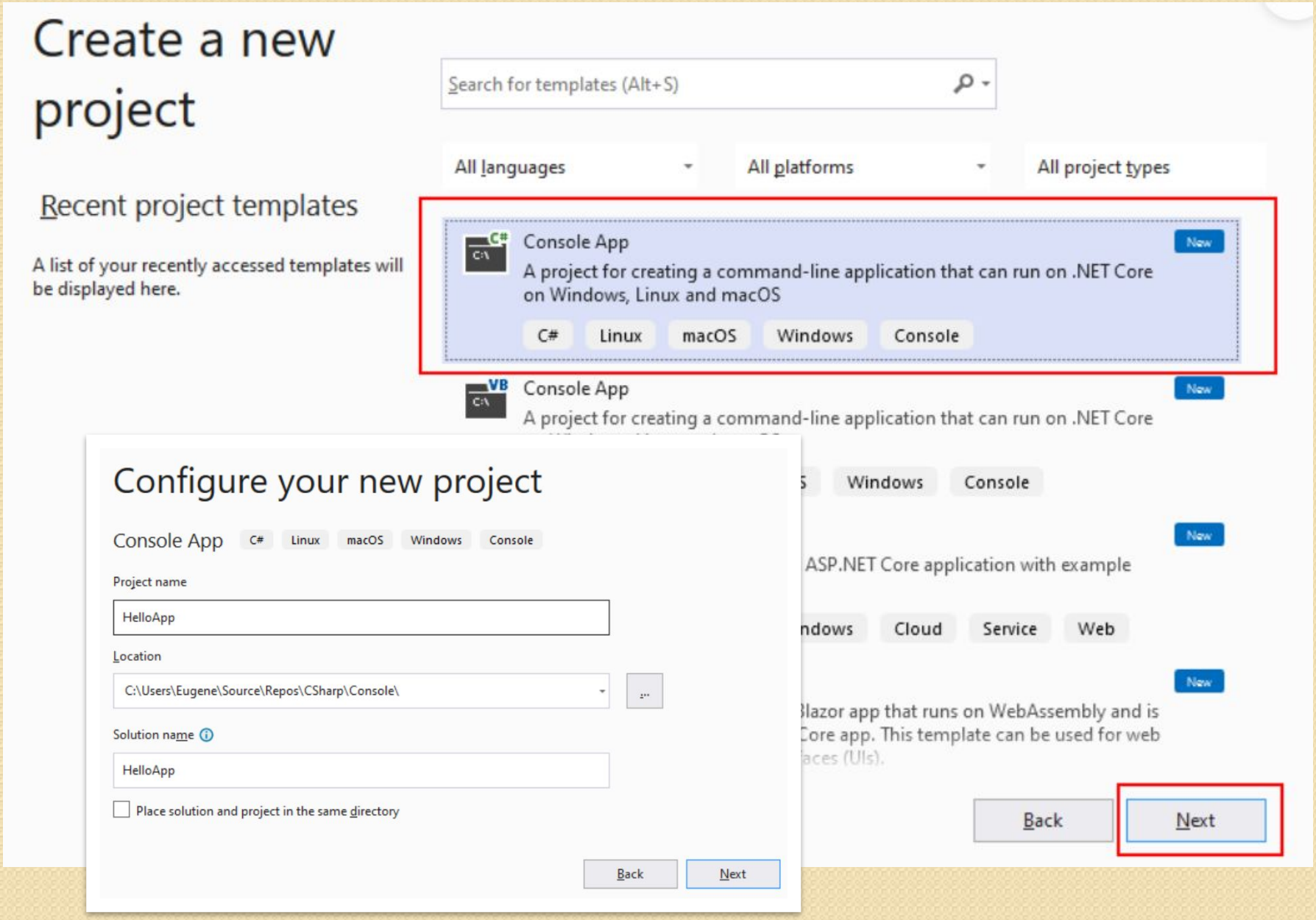
Фреймворк .NET представляет мощную платформу для создания приложений. Среди достоинств:

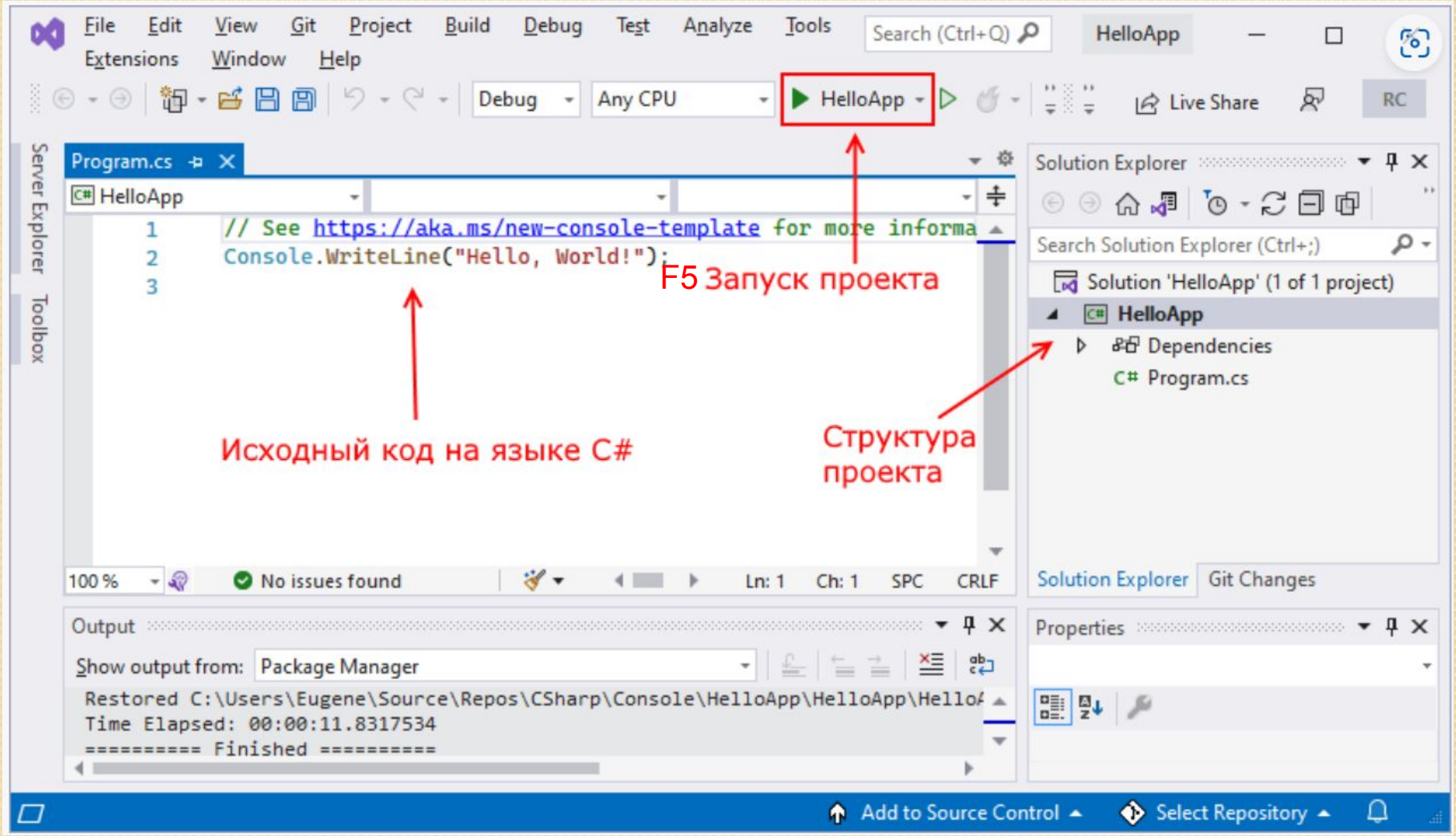
- ✔ Поддержка нескольких языков.
- ✔ Кроссплатформенность.
- ✔ Мощная библиотека классов.
- ✔ Разнообразие технологий.

Класс содержит *данные*, задающие свойства объектов класса, и *функции (методы)*, определяющие их поведение

- Вся логика заключена внутри класса.
- Вне класса работа не возможна.

ДЛЯ СОЗДАНИЯ ПРОЕКТА СЛЕДУЕТ ПОСЛЕ ЗАПУСКА VISUAL STUDIO.NET В ГЛАВНОМ МЕНЮ ВЫБРАТЬ КОМАНДУ FILE□ NEW PROJECT….





После запуска приложения его можете найти на жестком диске в папке проекта в каталоге **bin\Debug\net6.0**. Оно будет называться по имени проекта и иметь расширение exe. Этот файл можно будет запускать без Visual Studio, а также переносить его на другие компьютеры, где установлен .NET 6.

СОСТАВ ЯЗЫКА

Символы:

буквы: A-Z, a-z, _, буквы нац. алфавитов

цифры: 0-9, A-F

спец. символы: +, *, {, …

пробельные символы

- **Лексемы**:
  - константы 2 0.11 “Вася”
  - имена Vasia a _11
  - ключевые слова double do if
  - знаки операций + - =
  - разделители ; [ ] ,

- **Выражения**
  - выражение - правило вычисления значения: a + b
- **Операторы**
  - исполняемые: c = a + b;
  - описания: double a, b;

ИМЕНА (ИДЕНТИФИКАТОРЫ)

- имя должно начинаться с буквы или _;
- имя должно содержать только буквы, знак подчеркивания и цифры;
- прописные и строчные буквы различаются;
- длина имени практически не ограничена.
- имена не должны совпадать с ключевыми словами, однако допускается: @if, @float…
- в именах можно использовать управляющие последовательности Unicode

Примеры правильных имен:

Vasia, Вася, _13, \u00F2\u01DD, @while.

Примеры неправильных имен:

2late, Big gig, Б#г

НОТАЦИИ

Понятные и согласованные между собой имена — основа хорошего стиля. Существует несколько *нотаций* — соглашений о правилах создания имен.

В C# для именования различных видов программных объектов чаще всего используются две нотации:

- *Нотация Паскаля* - каждое слово начинается с прописной буквы:
  - MaxLength, MyFuzzyShooshpanchik
- *Camel notation* - с прописной буквы начинается каждое слово, составляющее идентификатор, кроме первого:
  - maxLength, myFuzzyShooshpanchik

ПЕРЕМЕННЫЕ

Переменная — это именованная область памяти, в которой хранятся данные определенного типа.

Все переменные, используемые в программе, должны быть описаны.

Для каждой переменной задается ее *имя и тип*:

```
int       number;
float          x, y;
char      option;
```

Тип переменной выбирается исходя из диапазона и требуемой точности представления данных.

Неявная типизация

```
var hello = "Hell to
World";
var c = 20;
```

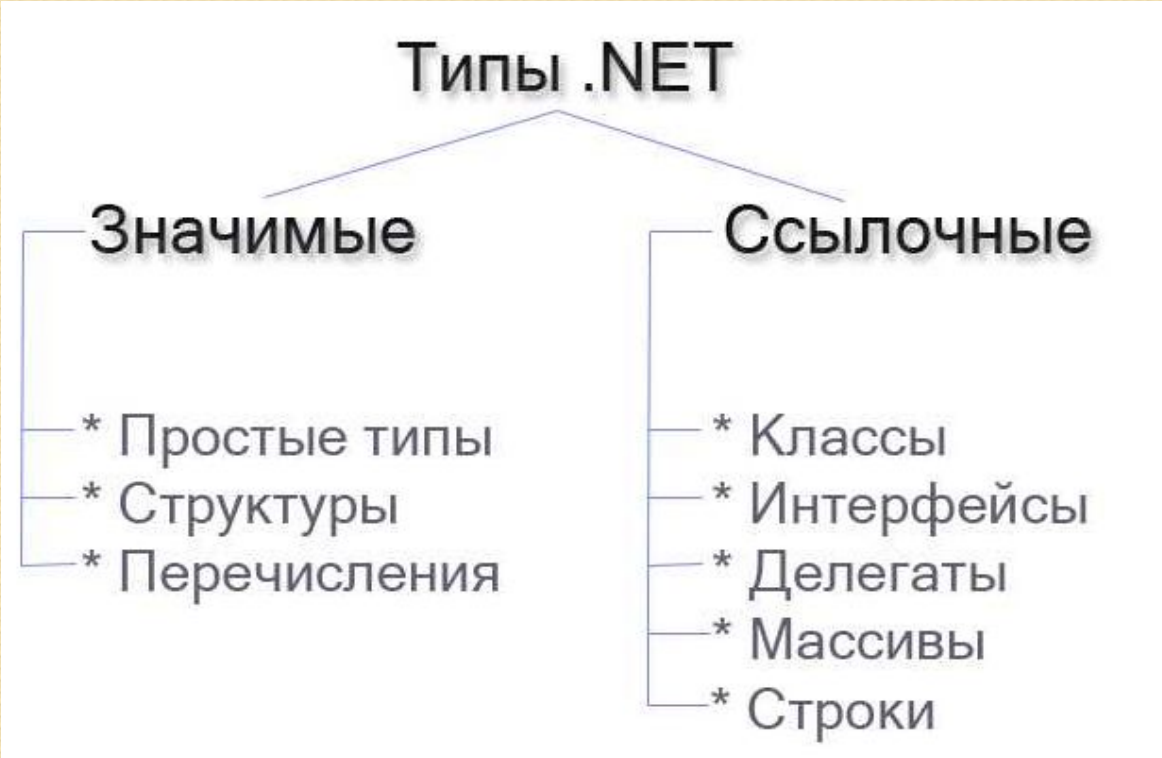
```
int a;  // этот код работает
a = 20;

var c;   // этот код не работает
c= 20;
```

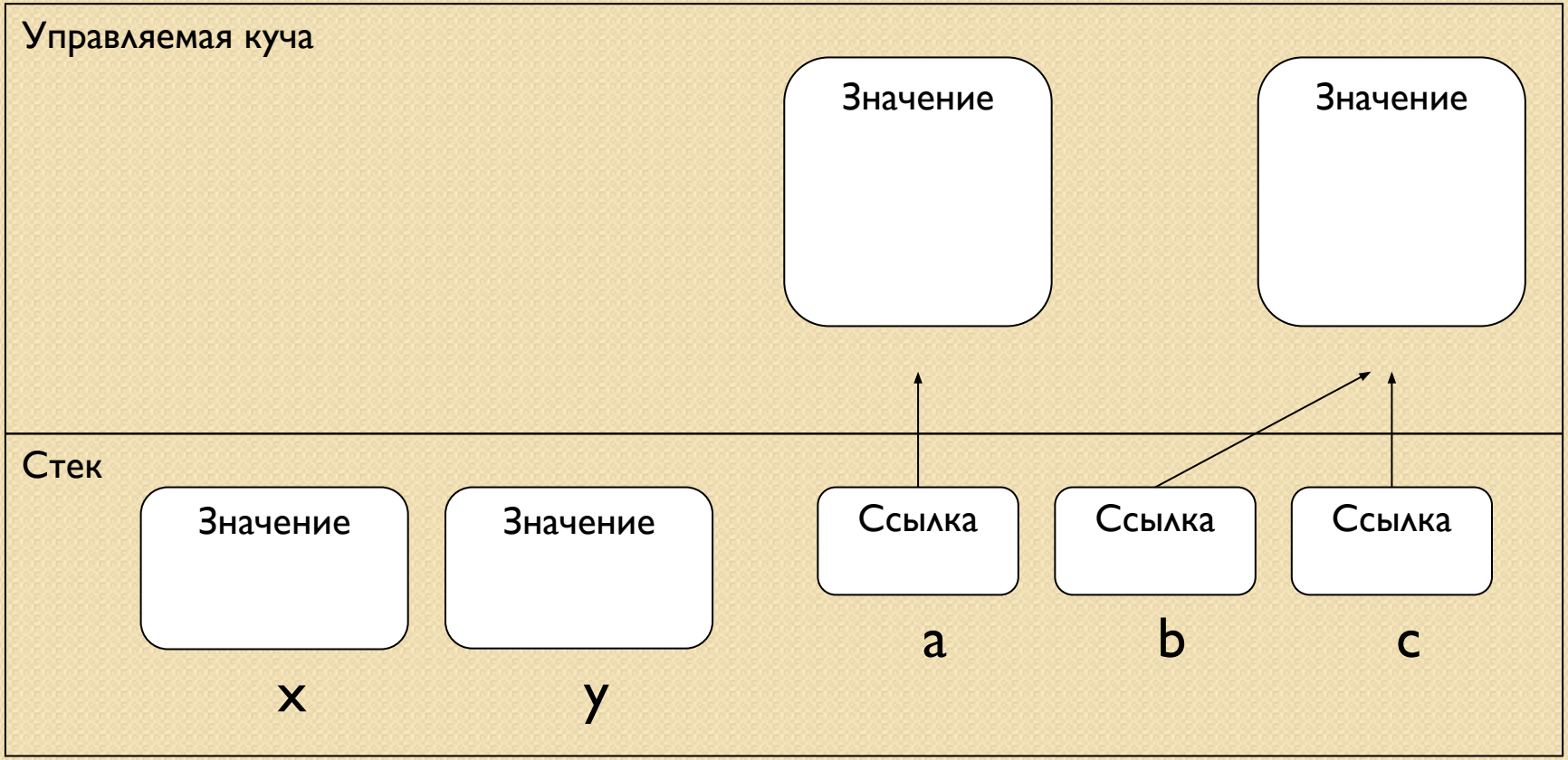
ТИПЫ ДАННЫХ

Тип данных определяет:

- внутреннее представление данных => *множество их возможных значений*
- допустимые действия над данными => *операции и функции*



ХРАНЕНИЕ В ПАМЯТИ ВЕЛИЧИН ЗНАЧИМОГО И ССЫЛОЧНОГО ТИПА



Тип-значение

Хранятся в стеке. Стек-это область памяти, которая активно используется процессором.

Ссылочный тип

Хранятся в управляемой куче (области оперативной памяти).

ЗНАЧИМЫЕ ТИПЫ

Логический тип			
Имя типа	**Системный тип**	**Значения**	**Размер**
Bool	System.Boolean	true, false	8 бит
Арифметические целочисленные типы			
Имя типа	**Системный тип**	**Диапазон**	**Размер**
Sbyte	System.SByte	-128 — 127	Знаковое, 8 Бит
Byte	System.Byte	0 — 255	Беззнаковое, 8 Бит
Short	System.Short	-32768 —32767	Знаковое, 16 Бит
Ushort	System.UShort	0 — 65535	Беззнаковое, 16 Бит
Int	System.Int32	≈(-2*10^9 — 2*10^9)	Знаковое, 32 Бит
Uint	System.UInt32	≈(0 — 4*10^9)	Беззнаковое, 32 Бит
Long	System.Int64	≈(-9*10^18 — 9*10^18)	Знаковое, 64 Бит
Ulong	System.UInt64	≈(0— 18*10^18)	Беззнаковое, 64 Бит
Арифметический тип с плавающей точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
Float	System.Single	+1.5*10^-45 - +3.4*10^38	7 цифр
Double	System.Double	+5.0*10^-324 - +1.7*10^308	15-16 цифр
Арифметический тип с фиксированной точкой			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
Decimal	System.Decimal	+1.0*10^-28 - +7.9*10^28	28-29 значащих цифр
Символьные типы			
Имя типа	**Системный тип**	**Диапазон**	**Точность**
Char	System.Char	U+0000 - U+ffff	16 бит Unicode символ
String	System.String	Строка из символов Unicode	
Объектный тип			
Имя типа	**Системный тип**	**Примечание**	
Object	System.Object	Прародитель всех *встроенных* и пользовательских типов	

ИНИЦИАЛИЗАЦИЯ ПЕРЕМЕННЫХ

- При объявлении можно присвоить переменной начальное значение (инициализировать).

```
int   number = 100;
float       x      = 0.02;
charoption = ’ю’;
```

При инициализации можно использовать не только константы, но и выражения — главное, чтобы на момент описания они были вычислимыми, например:

```
int b = 1, a = 100;
int x = b * a + 25;
```

- Инициализация локальных переменных возлагается на программиста. Рекомендуется всегда инициализировать переменные при описании.

```
int a, b, c;
a = b = c = 34;
```

ЛИТЕРАЛЫ

Литералы - это явно заданные значения в коде программы — константы определенного типа. Литералы можно передавать переменным в качестве значения.

Тип литерала	Пример значения
Логический	true (истина) и false (ложь)
Целочисленные	-11 0b11 (двоичное число 3) 0x0A (шестнадцатеричное число 10)
Вещественные	3.14 4d, 16D 7f
Символьные	'2' 'A' '\n' - перевод строки '\t' - табуляция '\x78' шестнадцатеричный код ASCII символа 'x' '\u0421' шестнадцатеричный код Unicode символа С
Строковые	"hello"

ОБЛАСТЬ ДЕЙСТВИЯ И ВРЕМЯ ЖИЗНИ ПЕРЕМЕННЫХ

- Переменные описываются внутри какого-л. блока (класса, метода или блока внутри метода)
  - **Блок** — это код, заключенный в фигурные скобки. Основное назначение блока — группировка операторов.
  - Переменные, описанные непосредственно внутри класса, называются **полями класса**.
  - Переменные, описанные внутри метода класса, называются **локальными переменными**.
- **Область действия переменной** - область программы, где можно использовать переменную.
- Область действия переменной начинается в точке ее описания и длится до конца блока, внутри которого она описана.
- **Время жизни**: переменные создаются при входе в их область действия (блок) и уничтожаются при выходе.

ИМЕНОВАННЫЕ КОНСТАНТЫ

Вместо значений констант можно (и нужно!) использовать в программе их имена.

Это облегчает читабельность программы и внесение в нее изменений:

```
const    float weight = 61.5;
const    int    n      = 10;
const    float  g       = 9.8;
```

ВЫРАЖЕНИЯ

- Выражение — правило вычисления значения.
- В выражении участвуют *операнды*, объединенные знаками операций.
- Операндами выражения могут быть константы, переменные и вызовы функций.
- Операции выполняются в соответствии с *приоритетами*.
- Для изменения порядка выполнения операций используются *круглые скобки*.
- Результатом выражения всегда является значение определенного типа, который определяется типами операндов.
- Величины, участвующие в выражении, должны быть *совместимых типов*.

- t + Math.Sin(x)/2 * x

результат имеет вещественный тип

- a <= b + 2

результат имеет логический тип

- x > 0 && y < 0

результат имеет логический тип

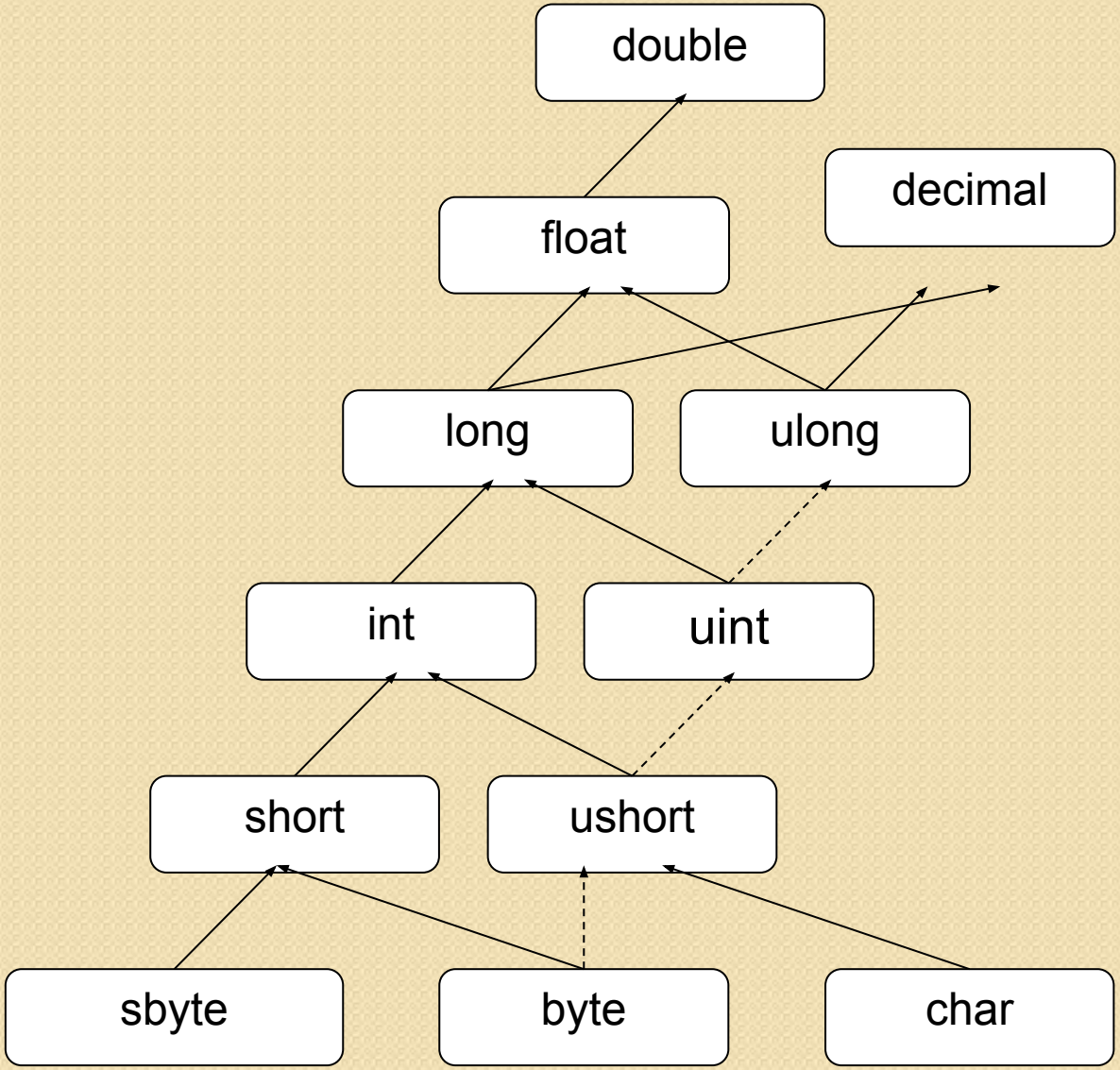
ПРИОРИТЕТЫ ОПЕРАЦИЙ C#

1. Первичные (), [], ++, --, new, …
2. Унарные ~, !, ++, --, -, …
3. Типа умножения (мультипликативные) *, /, %
4. Типа сложения (аддитивные) +, -
5. Сдвига <<, >>
6. Отношения и проверки типа <, >, is, …
7. Проверки на равенство ==, !=
8. Поразрядные логические &, ^, |
9. Условные логические &&, ||
10. Условная ?:
11. Присваивания =, *=, /=,…

ТИП РЕЗУЛЬТАТА ВЫРАЖЕНИЯ

- *Если операнды, входящие в выражение, одного типа, и операция для этого типа определена, то результат выражения будет иметь тот же тип.*
- Если операнды разного типа и (или) операция для этого типа не определена, перед вычислениями автоматически выполняется преобразование типа по правилам, обеспечивающим приведение более коротких типов к более длинным для сохранения значимости и точности.
- Автоматическое (*неявное*) преобразование возможно не всегда, а только если при этом не может случиться потеря значимости.
- Если неявного преобразования из одного типа в другой не существует, программист может задать *явное* преобразование типа с помощью операции:
  - (тип)x.
  - Convert.ToInt16(x)
  - Int16.Parse(строковая_переменная);

НЕЯВНЫЕ АРИФМЕТИЧЕСКИЕ ПРЕОБРАЗОВАНИЯ ТИПОВ В C#



ВЫВОД СООБЩЕНИЯ НА ЭКРАН

Console.Write(“текст сообщения") – вывод сообщения на экран.

Console.WriteLine(“текст сообщения") – вывод сообщения на экран и перенос каретки на следующую строку.

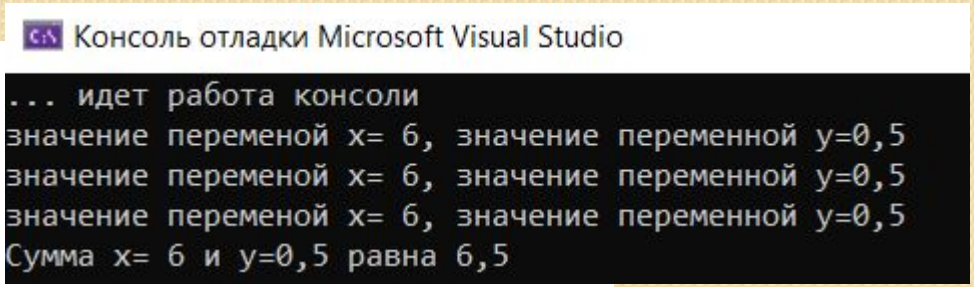
Console.WriteLine(“текст сообщения”+ x) – вывод на экран сообщения, заключенного в кавычки и значения переменной x и перенос каретки на следующую строку.

Console.WriteLine(“текст сообщения {0} {1}”, x,y) – вывод на экран сообщения, заключенного в кавычки и значения переменной x и y перенос курсора на следующую строку.

Console.WriteLine($“текст сообщения {x} {y}”) – вывод на экран сообщения, заключенного в кавычки и значения переменной x и y перенос курсора на следующую строку.

```
using static System.Console;

int x = 6;
double y = 0.5;
Console.WriteLine("... идет работа консоли");
WriteLine("значение переменой x= " + x + ", значение переменной y=" + y);
WriteLine("значение переменой x= {0}, значение переменной y={1}", x, y);
WriteLine($"значение переменой x= {x}, значение переменной y={y}");
WriteLine($"Сумма x= {x} и y={y} равна {x+y}");
```



ИНКРЕМЕНТ (++) И ДЕКРЕМЕНТ (--)

Существуют две формы рассматриваемых операций:
префиксная и *постфиксная*.

Если операторы записаны после переменной (x++ или x--) - это постфиксная форма.

При этом последовательно происходят следующие действия:

1. старое значение переменной сохраняется для использования в дальнейшем выражении, в котором встретилась эта переменная;
2. и только после этого ее значение сразу же изменяется на 1.

Если операторы записаны перед переменной (++x или –x) -это префиксная форма.

При этом последовательность действий такая:

1. вначале переменная изменяется на 1;
2. и только после этого используется в выражении.

ИНКРЕМЕНТ И ДЕКРЕМЕНТ

```
using System;
namespace CA1
{    class C1
   {    static void Main()
      {  int x = 3, y = 3;
         Console.Write( "Значение префиксного выражения: " );
         Console.WriteLine( ++x );
         Console.Write( "Значение x после приращения: " );
         Console.WriteLine( x );

         Console.Write( "Значение постфиксного выражения: " );
         Console.WriteLine( y++ );
         Console.Write( "Значение y после приращения: " );
         Console.WriteLine( y );
}}}
```

Результат работы программы:
Значение префиксного выражения: 4
Значение x после приращения: 4
Значение постфиксного выражения: 3
Значение y после приращения: 4

```
using System;
namespace ConsoleApplication1
{    class Program
   {    static void Main(string[] args)
      {
         int x=1,y=1;
         Console.WriteLine("x={0}, y= {1}", x,
    x * y + (x++)+x*y);
         x = 1; y = 1;
         Console.WriteLine("x={0}, y= {1}", x,
    x * y + (++x) + x * y);
         Console.ReadLine();
      }
   }
}
```

```
x=1, y= 4
x=1, y= 5
```

ОПЕРАЦИЯ ПРИСВАИВАНИЯ

Присваивание – это замена старого значения переменной на новое. Старое значение стирается бесследно.

Операция может использоваться в программе как законченный оператор.

переменная = выражение

```
a = b + c;
x = 1;
x = x + 0.5;
```

Сложное присваивание:

- x += 0.5; соответствует x = x + 0.5;
- x *= 0.5; соответствует x = x * 0.5;

- a %= 3; соответствует a = a % 3;
- a <<= 2; соответствует a = a << 2;

ОПЕРАЦИЯ УМНОЖЕНИЯ И ДЕЛЕНИЯ

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            int x = 11, y = 4;
            float z = 4;
            Console.WriteLine( z * y );          // Результат 16
            Console.WriteLine( x / y );          // Результат 2  (целочисленное деление)
            Console.WriteLine( x / z );          // Результат 2,75
            Console.WriteLine( x % y );          // Результат 3  (остаток)

}}}
```

ОПЕРАЦИИ СДВИГА

- *Операции сдвига* (<< и >>) применяются к целочисленным операндам. Они сдвигают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом.
- При *сдвиге влево* (<<) освободившиеся разряды обнуляются. При *сдвиге вправо* (>>) освободившиеся биты заполняются нулями, если первый операнд беззнакового типа, и знаковым разрядом в противном случае.
- Стандартные операции сдвига определены для типов int, uint, long и ulong.

ПРИМЕР ОПЕРАЦИИ СДВИГА

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            byte  a = 3, b = 9;
            sbyte c = 9, d = -9;
            Console.WriteLine( a << 1 );      // Результат 6
            Console.WriteLine( a << 2 );      // Результат 12
            Console.WriteLine( b >> 1 );      // Результат 4
            Console.WriteLine( c >> 1 );      // Результат 4
            Console.WriteLine( d >> 1 );      // Результат -5
        }
    }
}
```

ОПЕРАЦИИ ОТНОШЕНИЯ И ПРОВЕРКИ НА РАВЕНСТВО

- *Операции отношения* (<, <=, >, >=, ==, !=) сравнивают первый операнд со вторым.
- Операнды должны быть арифметического типа.
- Результат операции — логического типа, равен true или false.

x == y -- true, если x равно y, иначе false

x != y -- true, если x не равно y, иначе false

x < y -- true, если x меньше y, иначе false

x > y -- true, если x больше y, иначе false

x <= y -- true, если x меньше или равно y, иначе false

x >= y -- true, если x больше или равно y, иначе false

УСЛОВНЫЕ ЛОГИЧЕСКИЕ ОПЕРАЦИИ

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            Console.WriteLine( true && true  );      // Результат true
            Console.WriteLine( true && false );      // Результат false
            Console.WriteLine( true || true  );      // Результат true
            Console.WriteLine( true || false );      // Результат true
        }
    }
}
```

УСЛОВНАЯ ОПЕРАЦИЯ

- **операнд_1 ? операнд_2 : операнд_3**

Первый операнд — выражение, для которого существует неявное преобразование к логическому типу.

Если результат вычисления первого операнда равен true, то результатом будет значение второго операнда, иначе — третьего операнда.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            int a = 11, b = 4;
            int max = b > a ? b : a;
            Console.WriteLine( max );      // Результат 11
 }}}
```

ВВОД ДАННЫХ С КОНСОЛИ

```
using System;
namespace A
{    class Class1
     {    static void Main()
          {
               string s = Console.ReadLine();          // ввод строки

          char c = (char)Console.Read();          // ввод символа
          Console.ReadLine();

               string buf;                                  // буфер для ввода чисел
               buf = Console.ReadLine();
           int i = Convert.ToInt32( buf );          // преобразование в целое

           buf = Console.ReadLine();
           double x = Convert.ToDouble( buf ); // преобразование в вещ.

               buf = Console.ReadLine();
           double y = double.Parse( buf );        // преобразование в вещ.
          }
     }
}
```

МАТЕМАТИЧЕСКИЕ ФУНКЦИИ: КЛАСС MATH

Имя	Описание	Результат	Пояснения
Abs	Модуль	перегружен	|x| записывается как Abs(x)
Acos	Арккосинус	double	Acos(double x)
Asin	Арксинус	double	Asin(double x)
Atan	Арктангенс	double	Atan(double x)
Atan2	Арктангенс	double	Atan2(double x, double y)— угол, тангенс которого есть результат деления y на x
BigMul	Произведение	long	BigMul(int x, int y)
Ceiling	Округление до большего целого	double	Ceiling(double x)
Cos	Косинус	double	Cos(double x)
Cosh	Гиперболический косинус	double	Cosh(double x)
DivRem	Деление и остаток	перегружен	DivRem(x, y, rem)
E	База натурального логарифма (число *e*)	double	2,71828182845905
Exp	Экспонента	double	e^x записывается как Exp(x)

Floor	**Округление до меньшего целого**	**double**	**Floor(double x)**
IEEERemainder	**Остаток от деления**	**double**	**IEEERemainder(double x, double y)**
Log	**Натуральный логарифм**	**double**	**$\log_e x$ записывается как Log(x)**
Log10	**Десятичный логарифм**	**double**	**$\log_{10} x$ записывается как Log10(x)**
Max	**Максимум из двух чисел**	**перегружен**	**Max(x, y)**
Min	**Минимум из двух чисел**	**перегружен**	**Min(x, y)**
PI	**Значение числа π**	**double**	**3,14159265358979**
Pow	**Возведение в степень**	**double**	**x^y записывается как Pow(x, y)**
Round	**Округление**	**перегружен**	**Round(3.1) даст в результате 3 Round (3.8) даст в результате 4**
Sign	**Знак числа**	**int**	**аргументы перегружены**
Sin	**Синус**	**double**	**Sin(double x)**
Sinh	**гиперболический синус**	**double**	**Sinh(double x)**
Sqrt	**Квадратный корень**	**double**	**$\sqrt{x}$ записывается как Sqrt(x)**
Tan	**Тангенс**	**double**	**Tan(double x)**
Tanh	**Гиперболический тангенс**	**double**	**Tanh(double x)**

УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ ЯЗЫКА ВЫСОКОГО УРОВНЯ

Реализуют логику выполнения программы:

- следование
- ветвление
- цикл
- передача управления

БЛОК (СОСТАВНОЙ ОПЕРАТОР)

- Блок — это последовательность операторов, заключенная в операторные скобки:
  - begin end
  - { }
- Блок воспринимается компилятором как один оператор и может использоваться **всюду, где синтаксис требует одного оператора, а алгоритм — нескольких**.
- Блок может содержать один оператор или быть пустым.

ОПЕРАТОР «ВЫРАЖЕНИЕ»

Любое выражение, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении выражения.

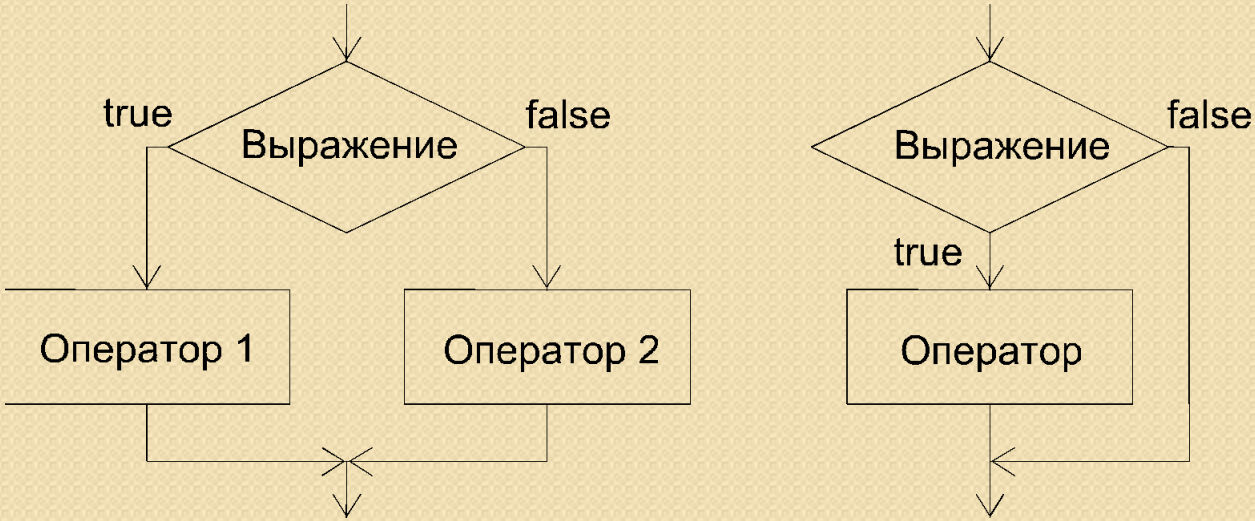
```
i++;              // выполняется операция инкремента
a *= b + c;       // выполняется умножение с присваиванием
fun( i, k );      // выполняется вызов функции
```

; называют пустым оператором.

Операторы ветвления:

- развилка (if)
- переключатель (switch)

УСЛОВНЫЙ ОПЕРАТОР IF



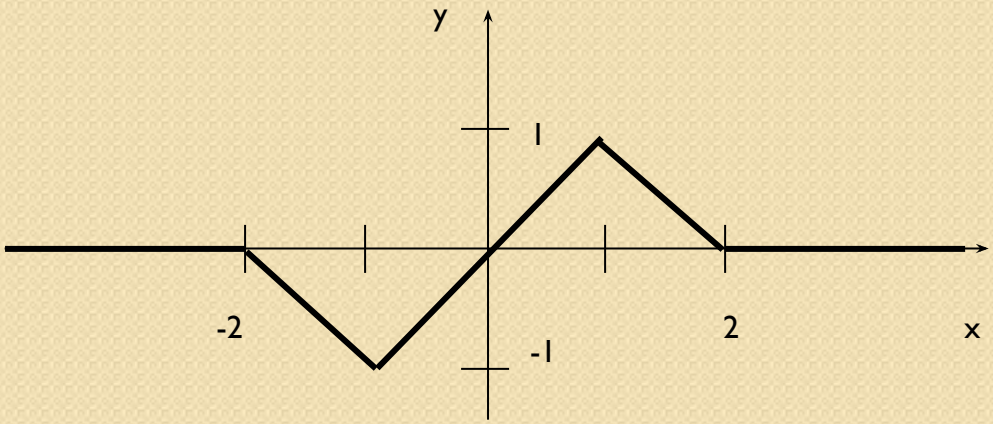
if (выражение) оператор_1; [else оператор_2;]

```
if ( a < 0 ) b = 1;


if ( a < b && (a > d || a = = 0))   ++b;
else {       b *= a; a = 0; }


if ( a < b ) if ( a < c ) m = a;
             else         m = c;
else         if ( b < c ) m = b;
             else         m = c;
```

ПРИМЕР



$$y = \begin{cases} 0, & x < -2 \\ -x - 2, & -2 \le x < -1 \\ x, & -1 \le x < 1 \\ -x + 2, & 1 \le x < 2 \\ 0, & x \ge 2 \end{cases}$$

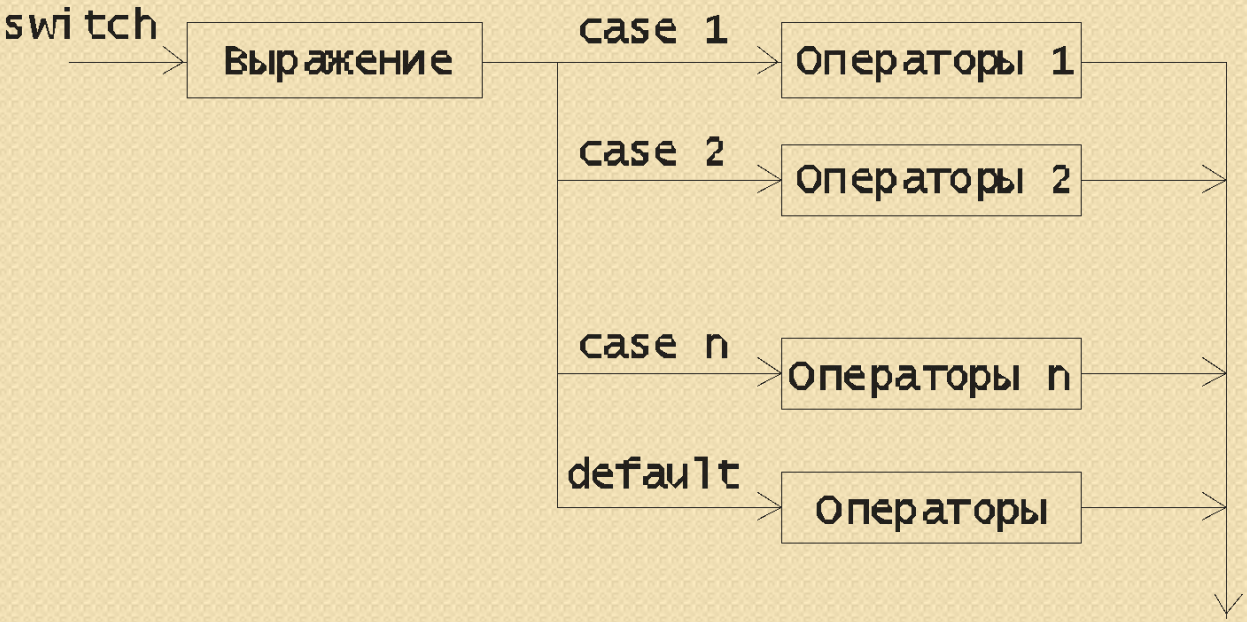
```
if ( x < -2 )                  y = 0;
if ( x >= -2 && x < -1 ) y = -x - 2;
if ( x >= -1 && x <  1 ) y = x;
if ( x >=  1 && x <  2 ) y = -x + 2;
if ( x >=  2 )                 y = 0;
```

```
if      ( x <= -2 )  y = 0;
else if ( x < -1 )     y = -x - 2;
  else if ( x <  1 )     y = x;
    else if ( x <  2 )     y = -x + 2;
    else y = 0;
```

```
y = 0;
if ( x > -2 ) y = -x - 2;
if ( x > -1 ) y = x;
if ( x >  1 ) y = -x + 2;
if ( x >  2 ) y = 0;
```

ОПЕРАТОР ВЫБОРА SWITCH

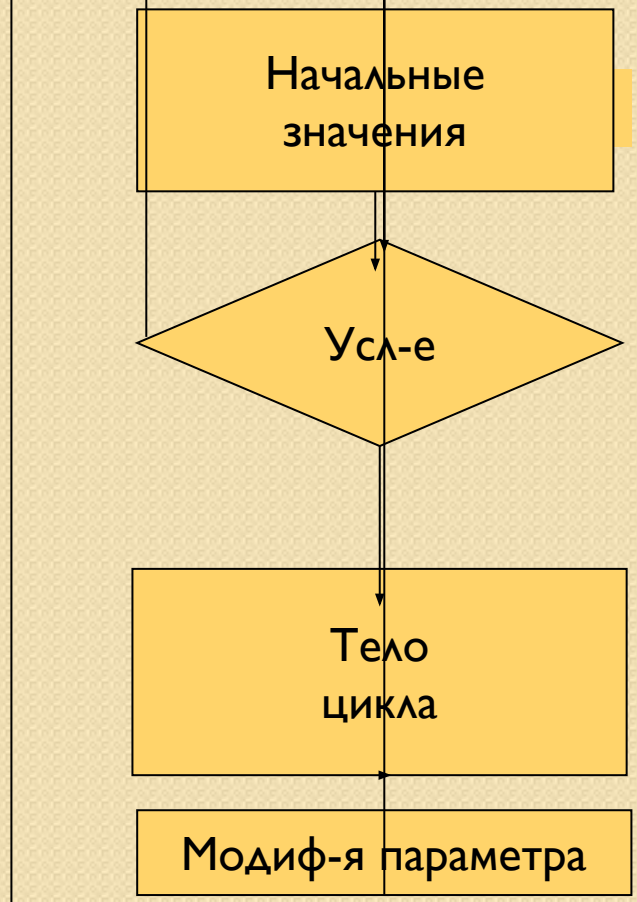
```
switch ( выражение )
{
    case константное_выражение_1: [ список_операторов_1 ]
    case константное_выражение_2: [ список_операторов_2 ]
    ..
    case константное_выражение_n: [ список_операторов_n ]
    [ default: операторы ]
}
```

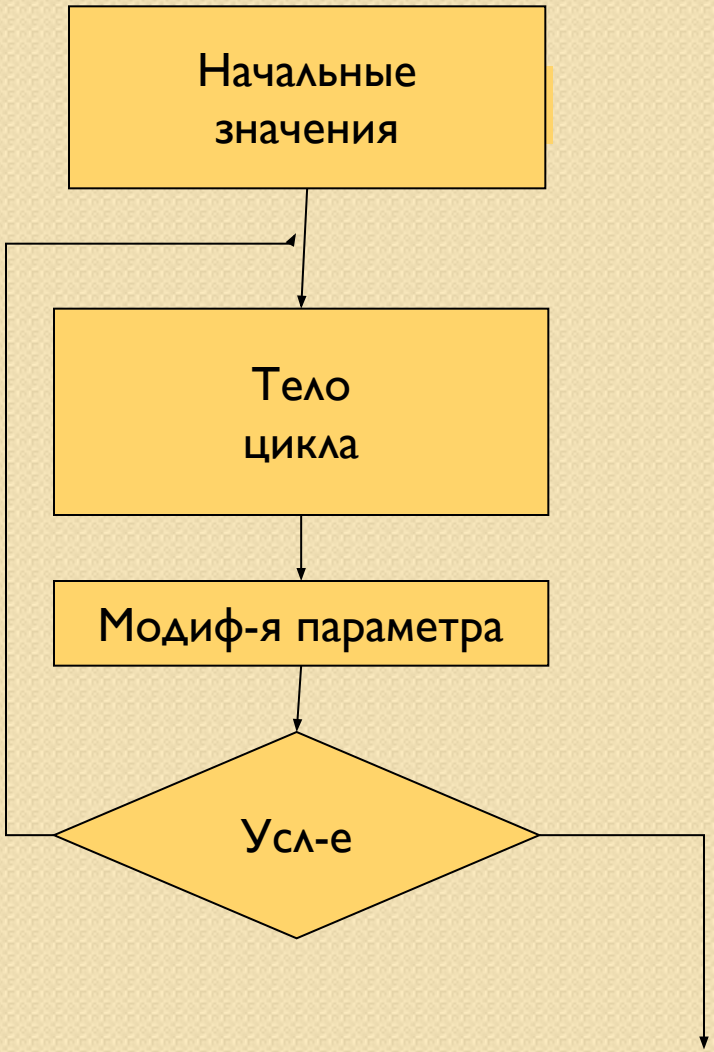


ПРИМЕР: КАЛЬКУЛЯТОР НА ЧЕТЫРЕ ДЕЙСТВИЯ

```
using System; namespace ConsoleApplication1
{   class Class1    {   static void Main()       {
        string buf;  double a, b, res;
          Console.WriteLine( "Введите 1й операнд:" );
        buf = Console.ReadLine(); a = double.Parse( buf);
        Console.WriteLine( "Введите знак операции" );
        char op = (char)Console.Read(); Console.ReadLine();
        Console.WriteLine( "Введите 2й операнд:" );
        buf = Console.ReadLine(); b = double.Parse( buf);
        bool ok = true;
        switch (op)
            {
                case '+' : res = a + b; break;
                case '-'  : res = a - b;  break;
                case '*' : res = a * b; break;
                case '/' : res = a / b; break;
                default  : res = double.NaN; ok = false; break;
            }
        if (ok) Console.WriteLine( "Результат: " + res );
         else    Console.WriteLine( "Недопустимая операция" );
}}}
```

СТРУКТУРА ОПЕРАТОРА ЦИКЛА

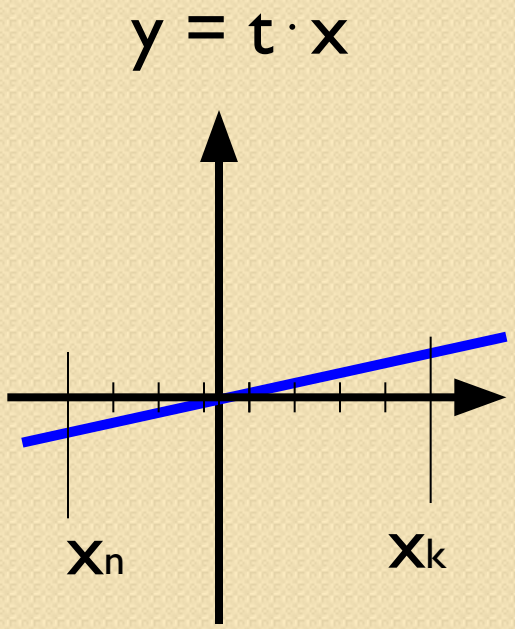




ЦИКЛ С ПРЕДУСЛОВИЕМ

while (выражение) оператор

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double Xn = -2, Xk = 12, dX = 2, t = 2, y;
            Console.WriteLine( "|     x     |     y     |" );
            double x = Xn;
            while ( x <= Xk )
            {
                y = t * x;
                Console.WriteLine( "| {0,9} | {1,9} |", x, y );
                x += dX;
            }
        }
    }
}
```



ЦИКЛ С ПОСТУСЛОВИЕМ

do оператор while выражение;

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            char answer;
            do
            {
                Console.WriteLine( "Купи слоника, а?" );
                answer = (char) Console.Read();
                Console.ReadLine();
            } while ( answer != 'y' );
        }
    }
}
```

ЦИКЛ С ПАРАМЕТРОМ

for (инициализация; выражение; модификации) оператор;

```
int s = 0;
for ( int i = 1; i <= 100; i++ ) s += i;
```

```
for ([действия_до_выполнения_цикла]; [условие]; [действия_после_выполнения])
{
    // действия
}
```

```
var i = 1;
for (WriteLine("Начало выполнения цикла"); i < 4; WriteLine($"i = {i}"))
{
    i++;
}
```

```
Начало выполнения цикла
i = 2
i = 3
i = 4
```

ПРИМЕР ЦИКЛА С ПАРАМЕТРОМ

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double Xn = -2, Xk = 12, dX = 2, t = 2, y;
            Console.WriteLine( "|     x     |     y     |";
            for ( double x = Xn; x <= Xk; x += dX )
            {
                y = t * x;
                Console.WriteLine( "| {0,9} | {1,9} |", x, y );
            }
        }
    }
}
```

РЕКОМЕНДАЦИИ ПО НАПИСАНИЮ ЦИКЛОВ

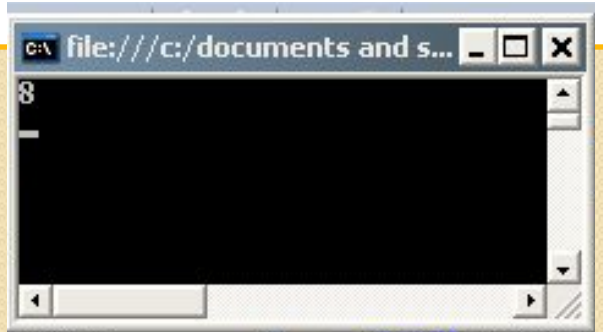
- не забывать о том, что если в теле циклов while и for требуется выполнить более одного оператора, нужно заключать их в блок;
- убедиться, что всем переменным, встречающимся в правой части операторов присваивания в теле цикла, до этого присвоены значения, а также возможно ли выполнение других операторов;
- проверить, изменяется ли в теле цикла хотя бы одна переменная, входящая в условие продолжения цикла;
- предусматривать аварийный выход из итеративного цикла по достижению некоторого предельно допустимого количества итераций.

ПЕРЕДАЧА УПРАВЛЕНИЯ

- оператор break — завершает выполнение цикла, внутри которого записан;
- оператор continue — выполняет переход к следующей итерации цикла;
- оператор return — выполняет выход из функции, внутри которой он записан;
- оператор throw — генерирует исключительную ситуацию;
- оператор goto — выполняет безусловную передачу управления.

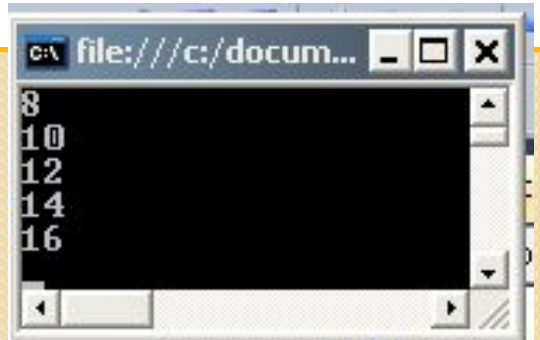
ОПЕРАТОР BREAK

```
using System;
namespace ConsoleApplication43
{   class Program
    {
        static void Main(string[] args)
        {
            int me = 8;
            for (int i = 0; i < 9; i++)
            {
                if (me % 2 == 0)
                {
                    Console.WriteLine(me);
                    me += 1;
                }
                else
                {
                    me += 1;
                    break;
                } }
            Console.ReadLine();
        }  } }
```



ОПЕРАТОР CONTINUE

```
using System;
namespace ConsoleApplication43
{   class Program
    {
        static void Main(string[] args)
        {
            int me = 8;
            for (int i = 0; i < 9; i++)
            {
                if (me % 2 == 0)
                {
                    Console.WriteLine(me);
                    me += 1;
                }
                else
                {
                    me += 1;
                    continue;
                } }
            Console.ReadLine();
        }  } }
```



ОПЕРАТОР GOTO

завершает выполнение функции и передает управление в точку ее вызова:

return [выражение];

Оператор goto

goto метка;

В теле той же функции должна присутствовать ровно одна конструкция вида:

метка: оператор;

goto case константное_выражение;

goto default;

ОПЕРАТОР GOTO: 2 ВАРИАНТА ИСПОЛЬЗОВАНИЯ

```
using System;
namespace ConsoleApplication43
{    class Program    {
      static void Main(string[] args)
      {
do
         {
            Console.WriteLine("введите слово или
символ 'x' для завершения работы приложения");
         str = Console.ReadLine();
            Console.WriteLine("вы ввели следующую
последовательность символов: {0}", str);
         if (str == "x") goto Finish;
         }
         while (true);

      Finish:
      Console.WriteLine("Нажмите клавишу ENTER,
чтобы завершить работу приложения");

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

```
using System;
namespace ConsoleApplication43
{    class Program    {
       static void Main(string[] args)
       { Console.WriteLine("Coffee sizes: 1=Small 2=Medium
3=Large");
          Console.Write("Please enter your selection: ");
          string s = Console.ReadLine();
          int n = int.Parse(s);   int cost = 0;
          switch (n)
          {     case 1:
                cost += 25;
                break;
             case 2:
                cost += 25;
                goto case 1;
             case 3:
                cost += 50;
                goto case 1;
             default:
                Console.WriteLine("Invalid selection.");
                break;
          }
          if (cost != 0) Console.WriteLine("Please insert {0}
cents.", cost);
Console.WriteLine("Thank you for your business.");
Console.WriteLine("Press any key to exit.");
          Console.ReadKey();      } } }
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ

- Исключительная ситуация, или исключение — это возникновение непредвиденного или аварийного события, которое может порождаться некорректным использованием аппаратуры.
- Например, это деление на ноль или обращение по несуществующему адресу памяти.

- Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработка.

ВОЗМОЖНЫЕ ДЕЙСТВИЯ ПРИ ОШИБКЕ

- прервать выполнение программы;
- возвратить значение, означающее «ошибка»;
- вывести сообщение об ошибке и вернуть вызывающей программе некоторое приемлемое значение, которое позволит ей продолжать работу;
- выбросить исключение

Исключения генерирует либо система выполнения, либо программист с помощью оператора throw.

НЕКОТОРЫЕ СТАНДАРТНЫЕ ИСКЛЮЧЕНИЯ

Имя	Пояснение
ArithmeticException	Ошибка в арифметических операциях или преобразованиях (является предком DivideBeZeroException и OverFlowException)
DivideByZeroException	Попытка деления на ноль
FormatException	Попытка передать в метод аргумент неверного формата
IndexOutOfRangeException	Индекс массива выходит за границы диапазона
InvalidCastException	Ошибка преобразования типа
OutOfMemoryException	Недостаточно памяти для создания нового объекта
OverFlowException	Переполнение при выполнении арифметических операций
StackOverFlowException	Переполнение стека

ОПЕРАТОР TRY

Служит для обнаружения и обработки исключений.

Оператор содержит три части:

- *контролируемый блок* — составной оператор, предваряемый ключевым словом try. В контролируемый блок включаются потенциально опасные операторы программы. Все функции, прямо или косвенно вызываемые из блока, также считаются ему принадлежащими;
- один или несколько *обработчиков исключений* — блоков catch, в которых описывается, как обрабатываются ошибки различных типов;
- *блок завершения* finally, выполняемый независимо от того, возникла ли ошибка в контролируемом блоке.

Синтаксис оператора try:

try блок [catch-блоки] [finally-блок]

МЕХАНИЗМ ОБРАБОТКИ ИСКЛЮЧЕНИЙ

- Обработка исключения начинается с появления ошибки. Функция или операция, в которой возникла ошибка, генерируют исключение;
- Выполнение текущего блока прекращается, отыскивается соответствующий обработчик исключения, и ему передается управление.
- В любом случае (была ошибка или нет) выполняется блок finally, если он присутствует.
- Если обработчик не найден, вызывается стандартный обработчик исключения.

ПРИМЕР РАБОТЫ ОПЕРАТОРА ОБРАБОТКИ ИСКЛЮЧЕНИЙ

```
using System;
namespace ConsoleApplication43
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Введите напряжение:");
                double u = double.Parse(Console.ReadLine());
                Console.WriteLine("Введите сопротивление:");
                double r = double.Parse(Console.ReadLine());
                double i = u / r;
                Console.WriteLine("Сила тока = " + i);
            }
            catch (FormatException)
                        Console.WriteLine("Неверный формат ввода!");
             catch                                          // общий случай
                        Console.WriteLine("Неопознанное исключение");
               finally
               Console.WriteLine("Нажмите любую клавишу для завершения работы программы");
              Console.ReadKey();
        }   } }
```

Массивы

- *Массив* — ограниченная совокупность однотипных величин
- Элементы массива имеют одно и то же имя, а различаются по порядковому номеру (*индексу*)

Создание массива

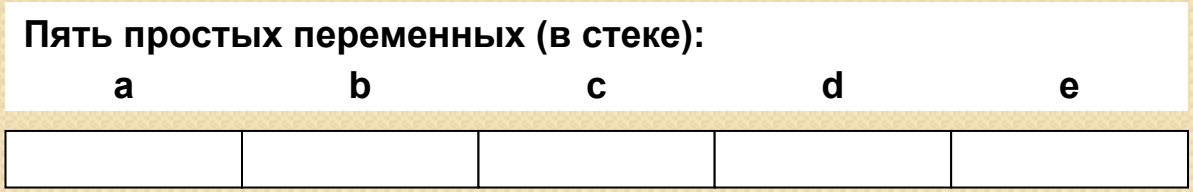
- **Массив относится к ссылочным типам данных** (располагается в управляемой куче), поэтому *создание массива* начинается с выделения памяти под его элементы.
- *Элементами массива* могут быть величины как значимых, так и ссылочных типов (в том числе массивы), например:

- **int[] w = new int[10];** // массив из 10 целых чисел
- **string[] z = new string[100];** // массив из 100 строк

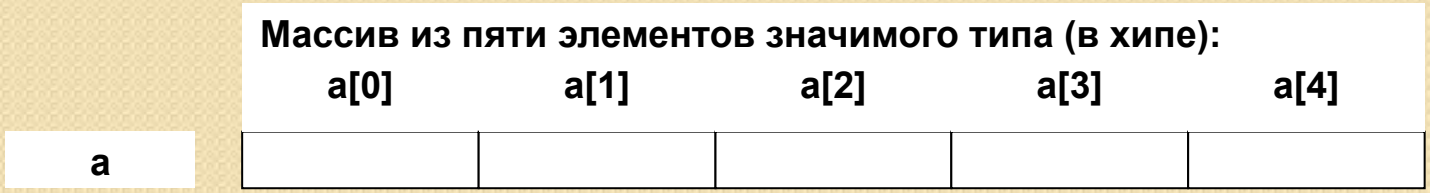
- Массив значимых типов хранит значения, массив ссылочных типов — ссылки на элементы.
- Всем элементам при создании массива присваиваются *значения по умолчанию*: нули для значимых типов и null для ссылочных.

Размещение массивов в памяти

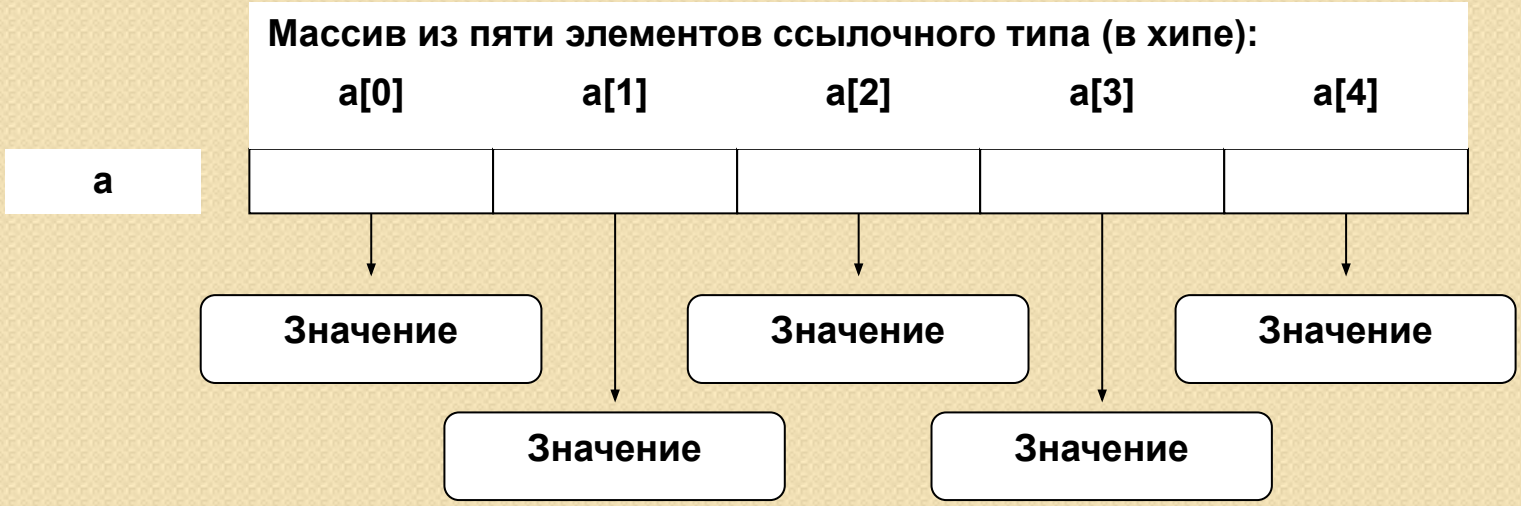
Пять простых переменных (в стеке):



Массив из пяти элементов значимого типа (в хипе):



Массив из пяти элементов ссылочного типа (в хипе):



Размерность массива

- Количество элементов в массиве (*размерность*) задается при выделении памяти и не может быть изменена впоследствии. Она может задаваться выражением:
- **short n = ...;**
- **string[] z = new string[2*n + 1];**
- Размерность не является частью типа массива.
- Элементы массива нумеруются *с нуля*.

- Для *обращения к элементу массива* после имени массива указывается номер элемента в квадратных скобках, например:
- **w[4] z[i]**
- С элементом массива можно делать все, что допустимо для переменных того же типа.
- При работе с массивом автоматически выполняется *контроль выхода за его границы*: если значение индекса выходит за границы массива, генерируется исключение IndexOutOfRangeException.

Действия с массивами

- Массивы одного типа можно *присваивать* друг другу. При этом происходит присваивание ссылок, а не элементов:

```
int[] a = new int[10];
int[] b = a;        // b и a указывают на один и тот же массив
```

- Все массивы в C# имеют общий базовый класс Array, определенный в пространстве имен System. Некоторые элементы класса Array:
  - Length (Свойство) - Количество элементов массива (по всем размерностям)
  - BinarySearch (Статический метод) - Двоичный поиск в отсортированном массиве
  - IndexOf – (Статический метод) - Поиск первого вхождения элемента в одномерный массив
  - Sort (Статический метод) - Упорядочивание элементов одномерного массива

Одномерные массивы

- Варианты описания массива:
- **тип[] имя**;
- **тип[] имя = new тип [размерность]**;
- **тип[] имя = { список_инициализаторов }**;
- **тип[] имя = new тип [] { список_инициализаторов }**;
- **тип[] имя = new тип [размерность] { список_инициализаторов }**;
- Примеры описаний (один пример на каждый вариант описания, соответственно):
- int[] a; // элементов нет
- int[] b = new int[4]; // элементы равны 0
- int[] c = { 61, 2, 5, -9 }; // new подразумевается
- int[] d = new int[] { 61, 2, 5, -9 }; // размерность вычисляется
- int[] e = new int[4] { 61, 2, 5, -9 }; // избыточное описание

Программа

Для массива, состоящего из 6 целочисленных элементов, программа определяет:

- сумму и количество отрицательных элементов;
- максимальный элемент.

```
const int n = 6;
int[] a = new int[n] { 3, 12, 5, -9, 8, -4 };

Console.WriteLine( "Исходный массив:" );
for ( int i = 0; i < n; ++i ) Console.Write( "\t" + a[i] );
Console.WriteLine();

long sum = 0;             // сумма отрицательных элементов
int  num = 0;             // количество отрицательных элементов
for ( int i = 0; i < n; ++i )
      if ( a[i] < 0 ) {
        sum += a[i];  ++num;
      }
 Console.WriteLine( "Сумма  отрицательных = " + sum );
 Console.WriteLine( "Кол-во отрицательных = " + num );

 int max = a[0];           // максимальный элемент
 for ( int i = 0; i < n; ++i )
         if ( a[i] > max ) max = a[i];
 Console.WriteLine( "Максимальный элемент = " + max );
```

Оператор foreach

- Применяется для перебора элементов массива. Синтаксис:
- **foreach (тип** имя **in** имя_массива **) тело_цикла**
- *Имя* задает локальную по отношению к циклу переменную, которая будет по очереди принимать все значения из массива, например:

- int[] massiv = { 24, 50, 18, 3, 16, -7, 9, -1 };
- foreach (int **x** in massiv) Console.WriteLine(**x**);

Использование методов класса Array

```
static void Main()
        {
            int[] a =  { 24, 50, 18, 3, 16, -7, 9, -1 };
            Console.WriteLine( "Исходный массив:", a );
            for ( int i = 0; i < a.Length; ++i )
                Console.Write( "\t" + a[i] );
            Console.WriteLine();

            Console.WriteLine( Array.IndexOf( a, 18 ) );
            Array.Sort(a);     // Array.Sort(a, 1, 5);
            Console.WriteLine ( "Упорядоченный массив:", a );
            for ( int i = 0; i < a.Length; ++i )
                Console.Write( "\t" + a[i] );
            Console.WriteLine();
            Console.WriteLine( Array.BinarySearch( a, 18) );
            Array.Reverse(a);     // Array.Reverse(a, 2, 4);
}
```

Прямоугольные массивы

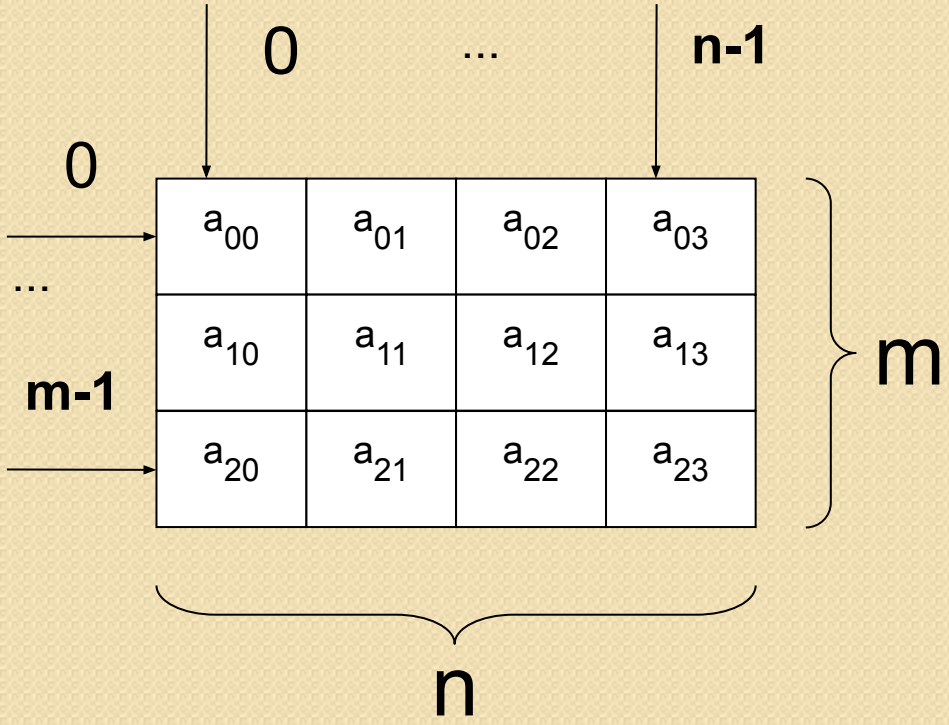
- *Прямоугольный массив* имеет более одного измерения. Чаще всего в программах используются двумерные массивы. Варианты описания двумерного массива:

- **тип[,] имя**;
- **тип[,] имя = new тип [разм_1**, **разм_2]**;
- **тип[,] имя = { список_инициализаторов }**;
- **тип[,] имя = new тип [,] { список_инициализаторов }**;
- **тип[,] имя = new тип [разм_1, разм_2] { список_инициализаторов }**;

- Примеры описаний (один пример на каждый вариант описания):

- int[,] a; // элементов нет
- int[,] b = new int[2, 3]; // элементы равны 0
- int[,] c = {{1, 2, 3}, {4, 5, 6}}; // new подразумевается
- int[,] c = new int[,] {{1, 2, 3}, {4, 5, 6}}; // разм-сть вычисляется
- int[,] d = new int[2,3] {{1, 2, 3}, {4, 5, 6}}; // избыточное описание

- К элементу двумерного массива обращаются, указывая номера строки и столбца, на пересечении которых он расположен:
- **a[1, 4] b[i, j] b[j, i]**
- Компилятор воспринимает как номер строки первый индекс, как бы он ни был обозначен в программе.



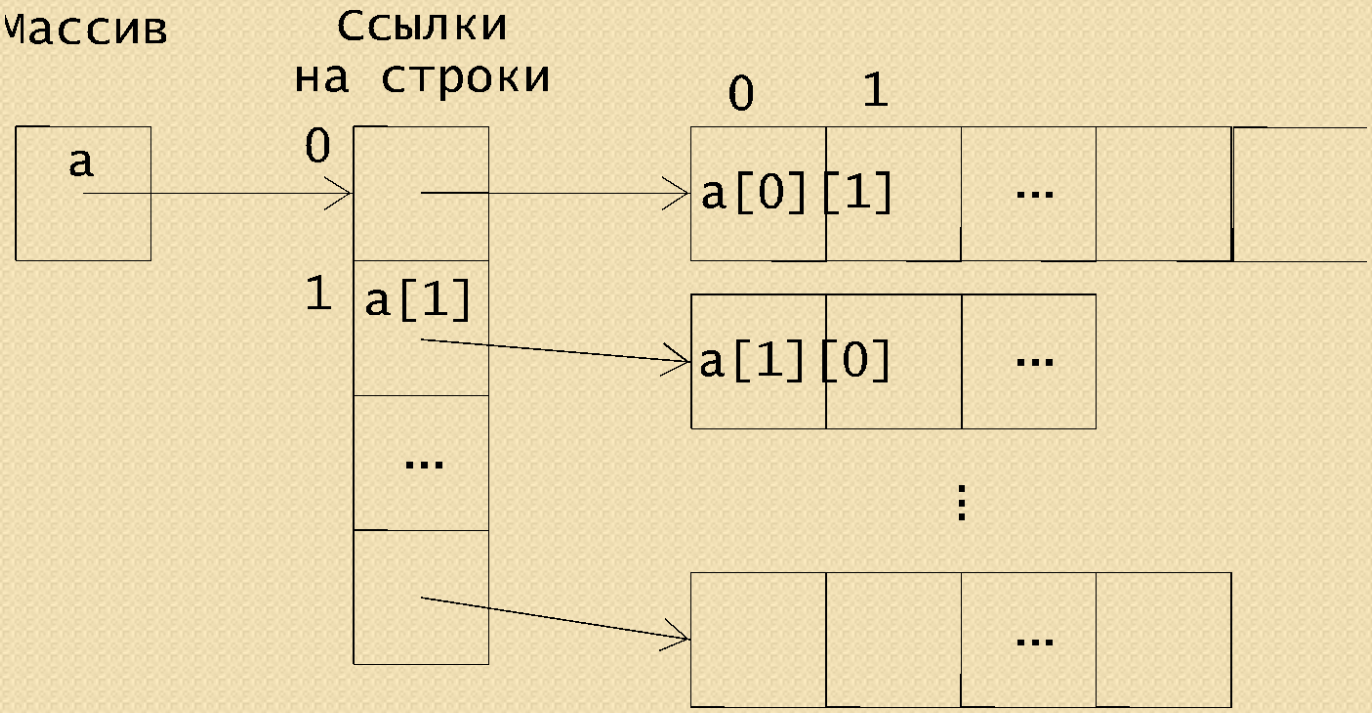

```
const int m = 3, n = 4;
int[,] a = new int[m, n] {
                { 2,-2, 8, 9 },
                {-4,-5, 6,-2 },
                { 7, 0, 1, 1 }
            };
Console.WriteLine( "Исходный массив:" );
for ( int i = 0; i < m; ++i )
{   for ( int j = 0; j < n; ++j )
        Console.Write( "\t" + a[i, j] );
    Console.WriteLine();
}
```

```
double sum = 0;
int nPosEl;
for ( int i = 0; i < m; ++i )
{
    nPosEl = 0;
    for ( int j = 0; j < n; ++j )
    {
        sum += a[i, j];
        if ( a[i, j] > 0 ) ++nPosEl;
    }
    Console.WriteLine( "В строке {0} {1} положит-х эл-в", i, nPosEl);
}
Console.WriteLine( "Среднее арифметическое всех элементов: "
                   + sum / m / n );
```

```
double sum = 0;
foreach ( int x in a ) sum += x; // все элементы двумерного массива!
Console.WriteLine( "Среднее арифметическое всех элементов: "
                    + sum / m / n );
```

Ступенчатые массивы

- В *ступенчатых массивах* количество элементов в разных строках может различаться. В памяти ступенчатый массив хранится иначе, чем прямоугольный: в виде нескольких внутренних массивов, каждый из которых имеет свой размер. Кроме того, выделяется отдельная область памяти для хранения ссылок на каждый из внутренних массивов.



Описание ступенчатого массива

- **тип**[][][] **имя**;

- Под каждый из массивов, составляющих ступенчатый массив, память требуется выделять явным образом:

- int[][] a = new int[3][]; // память под ссылки на 3 строки
- a[0] = new int[5]; // память под 0-ю строку (5 эл-в)
- a[1] = new int[3]; // память под 1-ю строку (3 эл-та)
- a[2] = new int[4]; // память под 2-ю строку (4 эл-та)

- Или:
- int[][] a = { new int[5], new int[3], new int[4] };

- Обращение к элементу ступенчатого массива:
- a[1][2] a[i][j] a[j][i]

Пример

```
int[][] a = new int[3][];
        a[0] = new int [5] { 24, 50, 18, 3, 16 };
        a[1] = new int [3] { 7, 9, -1 };
        a[2] = new int [4] { 6, 15, 3, 1 };
        Console.WriteLine( "Исходный массив:" );
        for ( int i = 0; i < a.Length; ++i )
          {
            for ( int j=0; j < a[i].Length; ++j)
                    Console.Write( "\t" + a[i][j] );
            Console.WriteLine();
          }
       // поиск числа 18 в нулевой строке:
       Console.WriteLine( Array.IndexOf( a[0], 18 ) );
```

```
foreach ( int [] mas1 in a )
  {
    foreach ( int x in mas1 )
          Console.Write( "\t" + x );
    Console.WriteLine();
  }
```

СТРОКИ В C#

- string
- StringBuilder

- символы (тип char)

СТРОКИ ТИПА STRING

Тип string предназначен для работы со строками символов в кодировке Unicode. Ему соответствует базовый класс System.String библиотеки .NET.

Создание строки:

1. string s; // инициализация отложена
2. string t = "qqq"; // инициализация строковым литералом
3. string u = new string(' ', 20); // с пом. конструктора
4. string v = new string(a); // создание из массива символов

// создание массива символов: char[] a = { '0', '0', '0' };

ОПЕРАЦИИ ДЛЯ СТРОК

- присваивание (=);
- проверка на равенство (==);
- проверка на неравенство (!=);
- обращение по индексу ([]);
- сцепление (конкатенация) строк (+).

- Строки равны, если имеют одинаковое количество символов и совпадают посимвольно.
- Обращаться к отдельному элементу строки по индексу можно только для получения значения, но не для его изменения. Это связано с тем, что строки типа string относятся к неизменяемым типам данных.
- Методы, изменяющие содержимое строки, на самом деле создают новую копию строки. Неиспользуемые «старые» копии автоматически удаляются сборщиком мусора.

НЕКОТОРЫЕ ЭЛЕМЕНТЫ КЛАССА SYSTEM.STRING

Название	Описание
Compare	Сравнение двух строк в лексикографическом (алфавитном) порядке. Разные реализации метода позволяют сравнивать строки и подстроки с учетом и без учета регистра и особенностей национального представления дат и т. д.
CompareOrdinal	Сравнение двух строк по кодам символов. Разные реализации метода позволяют сравнивать строки и подстроки
CompareTo	Сравнение текущего экземпляра строки с другой строкой
Concat	Конкатенация строк. Метод допускает сцепление произвольного числа строк
Copy	Создание копии строки

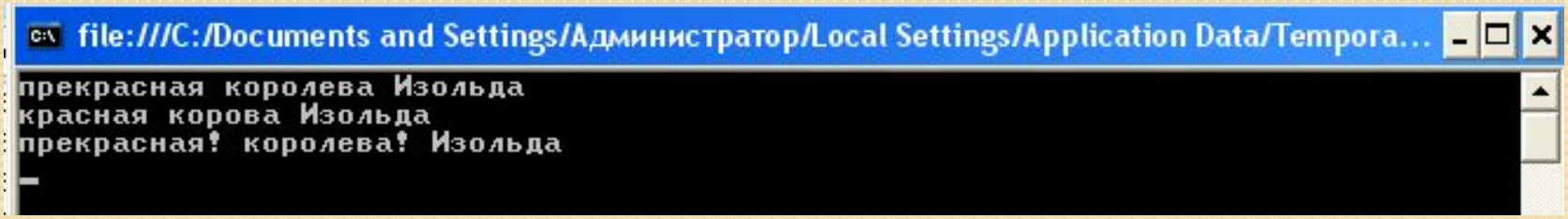
НЕКОТОРЫЕ ЭЛЕМЕНТЫ КЛАССА SYSTEM.STRING

Format	Форматирование в соответствии с заданными спецификаторами формата
IndexOf, LastIndexOf,…	Определение индексов первого и последнего вхождения заданной подстроки или любого символа из заданного набора
Insert	Вставка подстроки в заданную позицию
Join	Слияние массива строк в единую строку. Между элементами массива вставляются разделители (см. далее)
Length	Длина строки (количество символов)
Remove	Удаление подстроки из заданной позиции
Replace	Замена всех вхождений заданной подстроки или символа новой подстрокой или символом
Split	Разделение строки на элементы, используя заданные разделители. Результаты помещаются в массив строк
Substring	Выделение подстроки, начиная с заданной позиции

ПРИМЕР

```
string s = "прекрасная королева Изольда";
        Console.WriteLine( s );
        string sub = s.Substring( 3 ).Remove( 12, 2 );        // 1
        Console.WriteLine( sub );

        string[] mas = s.Split(' ');                          // 2
        string joined = string.Join( "! ", mas );
        Console.WriteLine( joined );
```



СТРОКИ ТИПА STRINGBUILDER

Класс StringBuilder определен в пространстве имен System.Text. Позволяет изменять значение своих экземпляров.

При создании экземпляра обязательно использовать операцию new и конструктор, например:

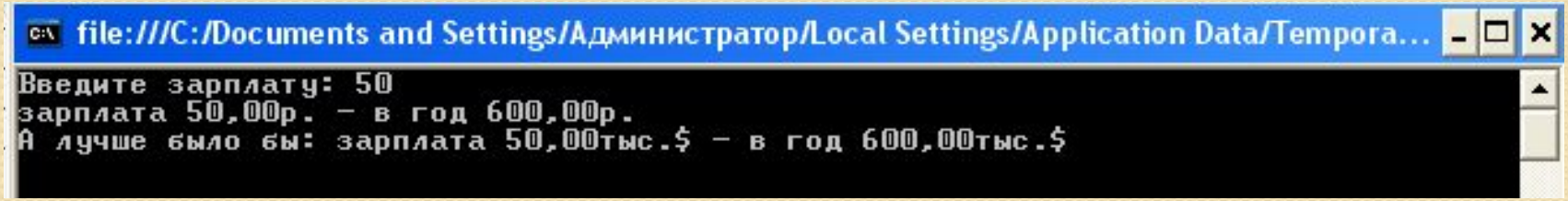
- StringBuilder a = new StringBuilder(); // 1
- StringBuilder b = new StringBuilder("qwerty"); // 2
- StringBuilder c = new StringBuilder(100); // 3
- StringBuilder d = new StringBuilder("qwerty", 100); // 4
- StringBuilder e = new StringBuilder("qwerty", 1, 3, 100);// 5

ОСНОВНЫЕ ЭЛЕМЕНТЫ КЛАССА SYSTEM.TEXT.STRINGBUILDER

Append	Добавление в конец строки. Разные варианты метода позволяют добавлять в строку величины любых встроенных типов, массивы символов, строки и подстроки типа string
AppendFormat	Добавление форматированной строки в конец строки
Capacity	Получение или установка емкости буфера. Если устанавливаемое значение меньше текущей длины строки или больше максимального, генерируется исключение ArgumentOutOfRangeException
Insert	Вставка подстроки в заданную позицию
Length	Длина строки (количество символов)
MaxCapacity	Максимальный размер буфера
Remove	Удаление подстроки из заданной позиции
Replace	Замена всех вхождений заданной подстроки или символа новой подстрокой или символом
ToString	Преобразование в строку типа string

ПРИМЕР

```
Console.Write( "Введите зарплату: " );
double   salary = double.Parse( Console.ReadLine());
StringBuilder a = new StringBuilder();
a.Append( "зарплата " );
a.AppendFormat( "{0, 6:C} - в год {1, 6:C}",  salary, salary * 12 );
Console.WriteLine( a );
a.Replace( "р.", "тыс.$" );
Console.WriteLine( "А лучше было бы: " + a );
```



СИМВОЛЬНЫЙ ТИП ДАННЫХ - CHAR

Символьный тип char предназначен для хранения символов в кодировке Unicode. Символьный тип относится к встроенным типам данных C# и соответствует стандартному классу Char библиотеки .NET из пространства имен System.

ОСНОВНЫЕ МЕТОДЫ

Метод	Описание
GetNumericValue	Возвращает числовое значение символа, если он является цифрой, и –1 в противном случае
IsControl	Возвращает true, если символ является управляющим
IsDigit	Возвращает true, если символ является десятичной цифрой
IsLetter	Возвращает true, если символ является буквой
IsLower	Возвращает true, если символ задан в нижнем регистре
IsUpper	Возвращает true, если символ записан в верхнем регистре
IsWhiteSpace	Возвращает true, если символ является пробельным (пробел, перевод строки и возврат каретки)
Parse	Преобразует строку в символ (строка должна состоять из одного символа)
ToLower	Преобразует символ в нижний регистр
MaxValue, MinValue	Возвращают символы с максимальным и минимальным кодами (эти символы не имеют видимого представления)

ПРИМЕР

```
namespace ConsoleApplication1
{  class Class1
   {  static void Main()
      {
        try
        {
          char b = 'B', c = '\x63', d = '\u0032';                    // 1
          Console.WriteLine( "{0}  {1}  {2}", b, c, d );
          Console.WriteLine( "{0}  {1}  {2}",
               char.ToLower(b), char.ToUpper(c), char.GetNumericValue(d) );

          char a;
          do                                                  // 2
          {
             Console.Write( "Введите символ: " );
             a = char.Parse( Console.ReadLine() );
             Console.WriteLine( "Введен символ {0}, его код – {1}",
                              a, (int)a );
             if (char.IsLetter(a))      Console.WriteLine("Буква");
             if (char.IsUpper(a))       Console.WriteLine("Верхний рег.");
             if (char.IsLower(a))       Console.WriteLine("Нижний рег.");
             if (char.IsControl(a))     Console.WriteLine("Управляющий");
             if (char.IsNumber(a))      Console.WriteLine("Число");
             if (char.IsPunctuation(a)) Console.WriteLine("Разделитель");
          } while (a != 'q');
        }
        catch
        {
          Console.WriteLine( "Возникло исключение" );
          return;
        } } } }
```