Основы программирования на C#

доцент кафедры Программирования и компьютерной техники
Киевского национального университета имени Тараса Шевченко
Бычков Алексей Сергеевич

Содержание

- Строки как объекты
- Массивы

Строки и символы

Символы в C# (System.Char) представлены символами Unicode и занимают 2 байта.
Символьный литерал берется в одиночные кавычки:

```
char c = 'A';       // инициализация символа
```

Кроме символьных литералов имеются еще Esc-последовательности. Например,

```
char newLine = '\n';                    // новая строка
char backSlash = '\\';                  // обратный слеш
```

Строки и символы

Esc-последовательности:

Esc-послед.	Символ	Значение
\'	Один.кавычка	0x0027
\"	Двойн.кавычка	0x0022
\\	Обратн.слеш	0x005C
\b	Backspace	0x0008
\n	Новая строка	0x000A
\r	Возврат каретки	0x000D
\t	Горизонтальный Tab	0x0009

Строки и символы

Последовательность **\u** предоставляет возможность записать любой символ **Unicode** через его шестнадцатеричное представление:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

Строки и символы

Допускается неявное преобразование символов в целые числа для типов, в которых может разместиться **ushort** . Для переменных типа **byte** и других числовых требуется явное преобразование.

```
char c = 'A';
    ushort x;
byte q;
     x = c;                  // неявное преобр.
q = (byte)c;                 // явное преобр.
    Console.WriteLine("c={0}",c);
    Console.WriteLine("x={0}",x);
Console.WriteLine("q={0}", q);
```

Строки и символы

Рассмотрим некоторые методы для работы с переменными типа **char:**

```
Console.WriteLine(char.IsDigit('1'));              // true, это цифра
Console.WriteLine(char.IsLetter('a'));             // true, это буква
Console.WriteLine(char.IsLetterOrDigit('!'));      // false, это и не
                                                   // буква, и не цифра
Console.WriteLine(char.IsPunctuation(','));        // true, это знак
                                                   // пунктуации
Console.WriteLine(char.IsWhiteSpace(' '));         // true, это
                                                   // разделитель
Console.WriteLine(char.IsUpper('B'));              // true, это буква
                                                   //верхнего регистра
Console.WriteLine(char.IsLower('b'));              // true, это буква
                                                   //нижнего регистра
```

Строки и символы

Строки в Си-шарп - это объекты класса **String**, значением которых является текст.
Чтобы использовать строку, ее нужно сначала создать и присвоить какое-либо значение, иначе мы получим ошибку:

"Использование локальной переменной "[имя переменной]", которой не присвоено значение".

Строки

Объявим простую строку и выведем ее на экран:

```
static void Main(string[] args)
{
   string s1 = "I'm a programmer!";
   Console.WriteLine(s1);
   string s2;
   Console.WriteLine(s2);      // ошибка, строка не инициализирована
}
```

Строки

Объявим простую строку и выведем ее на экран:

```
static void Main(string[] args)
{
   string s1 = "I'm a programmer!";
   Console.WriteLine(s1);
   string s2;
   Console.WriteLine(s2);      // ошибка, строка не инициализирована
}
```

Строки

- Тип **string** служит для представления последовательности текстовых символов. Каждый символ в такой последовательности относится к типу **char.** Это 16-разрядное значение, представляющее одну кодовую единицу в формате UTF**-16**.
- Строки в .NET являются неизменяемыми. Существует много операций, которые, казалось бы, должны модифицировать строку (например, конкатенация), но они создают новую строку, оставляя исходную без изменений.
- Как создавать изменяемые строки, мы рассмотрим позже.

Строки

Следует отметить, что тип **string** является ссылочным. Это означает, что в переменной этого типа хранится адрес участка памяти куче, где находится сама строка.

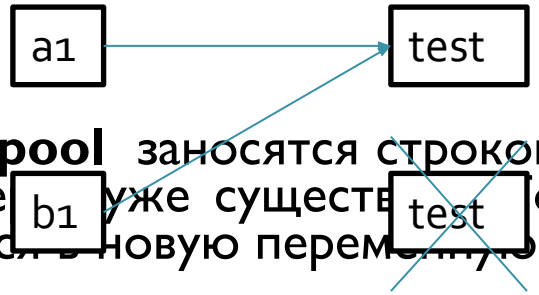
Но при этом, следующее сравнение «сравнивает значения»:

```
string a = "test";
string b = "test";
Console.WriteLine (a == b);                    //
```

Строки

Но как???

При объявлении и инициализации переменной **b1** C# проверяет так называемый **intern pool** (внутренний словарь строковых литералов).



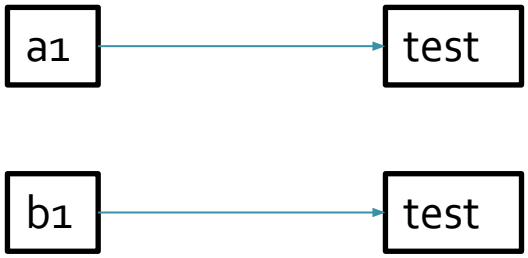
В **intern pool** заносятся строковые литералы при их создании. И если лите уже существ о новый не создается, а просто копируется в новую перем ю его адрес.

Строки

Посмотрите на следующий код:

```
string a1 = "test";
string b1 = String.Copy(a1);
```

Создается НОВЫЙ экземпляр типа string и поэтому **a1** и **b1** хранят разные адреса памяти, но по этим адресам расположены одинаковые наборы символов.



Строки

Тогда при выполнении следующего фрагмента

```
Console.WriteLine(a1 == b1);          // True
Console.WriteLine((object)a1 == (object)b1);
                                      // False
```

Имеем, что в первом случае сравниваются значения (т.е. test), а во втором – адреса, по которым эти значения расположены.

Строки

Что такое неизменяемость строки (в совокупности с ссылочным типом)?

Например имеем:

```
string s1 = "abcdef";
string s2 = s1;
s1[0] = 'z';              // невозможно!
```

Невозможно по причине того, что обе переменные **s1** и **s2** ссылаются на один и тот же участок памяти. И при изменении **s1 автоматически** «менялась бы» **s2!**

Строки

Следует аккуратно выполнять операторы сравнения со строками. Например, == и != можно использовать, а другие – нет.

Это связано с тем, в разных странах по-разному реализован лексикографический порядок.

Строки

Как уже отмечалось, с символа \ начинается Esc-последовательность. Поэтому, для хранения сроки вида

D:\project\Lecture 5

Необходимо записывать символ \ удвоенным:

```
string s1 = "D:\\project\\Lecture 5";
```

Строки

Однако, можно, используя символ @ избегать удвоения. Например:

```
string s2 = @"D:\\project\\Lecture 5 ";
```

Обратите внимание на отсутствие пробела после @!!!

Строки. Некоторые полезные операции и методы.

Конкатенация. Слияние строк.

```
static void Main(string[] args)
    {
string a = "I'm " + "programmer!";
string b = "I'm programmer!";
Console.WriteLine(a == b);        // True
    }
```

Строки. Некоторые полезные операции и методы.

Доступ. Взятие символа по номеру. Следует отметить, что символы в строке нумеруются, начиная с нуля!

```
static void Main(string[] args)
{
string b = "I'm programmer!";
char c = b[0];                                  // 'I'
Console.WriteLine("c={0}",c);
}
```

Строки. Некоторые полезные операции и методы.

Свойство **Length** возвращает длину строки

```
static void Main(string[] args)
{
string b = "I'm programmer!";
Console.WriteLine("длина={0}", b.Length);
}
```

Строки. Некоторые полезные операции и методы.

Метод **IsNullOrEmpty()** возвращает **True**, если значение строки равно **null**, либо когда она пуста (значение равно ""):

```
static void Main(string[] args)
{
   string s1 = null, s2 = "", s3 = "programmer";
      Console.WriteLine (String.IsNullOrEmpty(s1));    // True
      Console.WriteLine (String.IsNullOrEmpty(s2));    // True
      Console.WriteLine (String.IsNullOrEmpty(s3));    // False
}
```

Строки. Некоторые полезные операции и методы.

Метод **IsNullOrWhiteSpace()** возвращает True тогда, когда строка равна **null**, когда она пуста, либо строка представляет собой набор символов пробела и/или табуляции ("\t"):

```
static void Main(string[] args)
{
   string s1 = null, s2 = "\t", s3 = " ", s4 = "Hello";
   String.IsNullOrWhiteSpace(s1);                    // True
   String.IsNullOrWhiteSpace(s2);                 // True
   String.IsNullOrWhiteSpace(s3);                    // True
   String.IsNullOrWhiteSpace(s4);                    // False
}
```

Строки. Некоторые полезные операции и методы.

Сравнение строк. Строки сравниваются интересно. Например, строка "a" **меньше** строки "b", "bb" **больше** строки "ba".

Если обе строки равны - метод возвращает "0", если первая строка меньше второй – "-1", если первая больше второй "1":

```
static void Main(string[] args)
{
  Console.WriteLine(String.Compare("a", "b"));        // -1
  Console.WriteLine(String.Compare("a", "a"));        // 0
  Console.WriteLine(String.Compare("b", "a"));        // 1
  Console.WriteLine(String.Compare("ab", "abc"));     // -1
}
```

Строки. Некоторые полезные операции и методы.

Другие методы:

ToUpper() и **ToLower()**: переводит всю строку в верхний/нижний регистр.
Contains(): проверяет, содержит ли строка подстроку.
IndexOf(): возвращает индекс первого символа подстроки, которую содержит строка.
Insert(): используется для вставки подстроки в строку, начиная с указанной позиции.
Remove(): обрезает строку, начиная с некоторой позиции.
Substring(): получает подстроку из строки, начиная с указанной позиции.
Replace(): заменяет в строке все подстроки указанной новой подстрокой.
ToCharArray() возвращает массив символов указанной строки.
Split(): разбивает строку по указанному символу на массив подстрок.

Строки. Некоторые полезные операции и методы.

Значения, которые хранятся в строках можно записать в файл. C# хранит строки в кодировке Unicode. Для работы с различными кодировками (UTF-8, windows-1251 и др.) можно использовать инструменты из пространства имен System.Text. Для работы с файлами необходимо использовать методы из пространства имен System.IO.

```
using System;
using System.IO;
using System.Text;

string s = "Я программист";
// Сохраняем в файл в кодировке UTF-8 без BOM (специальной служебной информации о кодировке 2 байта
File.WriteAllText("default.txt", s); // из класса File пространства имен System.IO используем метод WriteAllText
        // Сохраняем в Windows-1251.
File.WriteAllText("windows-1251.txt", s, Encoding.GetEncoding(1251)); // из класса Encoding пространства имен System.Text используем метод GetEncoding

         // Сохраняем в UTF-8.
        File.WriteAllText("utf-8.txt", s, Encoding.UTF8);

        // Ожидание...
        Console.ReadKey();
```

Массивы.

Массив – это набор однотипных пронумерованных данных, которые располагаются в памяти последовательно друг за другом. **Нумерация начинается с нуля!**

Доступ к элементам **массива** осуществляется по индексу (номеру) элемента.
Массив может содержать элементы любого типа данных, можно создавать массив массивов (ступенчатый массив).
Количество элементов в массиве называется **размером** массива.
Массивы относятся к ссылочным типам данных.
Массивы в C# могут быть одномерными и многомерными.
Количество индексов у элемента массива называется **размерностью**.

Массивы.

В C# объявление массива имеет такой вид:

тип[] имя_массива = new тип[размер массива];

Например:

int[] array = new int[5]; // объявление и создание массива целых чисел
string[] colours = new string[4] {“blue",“red",“black",“grey"}; // объявление и создание массива цветов

Если при объявлении массива выполняется инициализация, оператор **new** можно упускать:
string[] colors = {“blue",“red",“black",“grey"};

Массивы.

В C# возможно неявное объявление массива. Ключевое слово **var** позволяет определить массив так, чтобы лежащий в ее основе тип задавался компилятором.

```
var myArray1 = new[] { 1, 2, 3 };
Console.WriteLine("Тип массива myArray1 - {0}",
myArray1.GetType());
var myArray2 = new[] { "One", "Two", "Three" };
Console.WriteLine("Тип массива myArray2 - {0}",
myArray2.GetType());
```

Массивы.

Следует напомнить, что C# в основе каждого типа (в системе типов .NET) лежит базовый класс **System.Object**. Поэтому можно определить массив, состоящий из элементов различного типа:

```
object[] myArrayByObject = { true, 10, "Привет", pi=3.14};
// печать типа каждого члена массива
foreach (object el in myArrayByObject)
Console.WriteLine("Тип {0} - {1}",el, el.GetType());
```

Массивы.

Доступ к элементам осуществляется с помощью указания номера элемента - индекса. Индексация начинается с нуля – первый элемент массива имеет индекс 0, а последний n-1, где **n** – количество элементов массива, его размер.

```
static void Main(string[] args)
{
   int[] numbers = new int[5];
   numbers[0] = 5;
   numbers[1] = 2;
   numbers[4] = 3;
   numbers[5] = 2; // ошибка, выход за пределы массива в памяти
}
```

Массивы.

Для хранения, например, табличных данных удобно использовать частный случай многомерных массивов – двухмерный.

В таком случае для доступа к элементу необходимо указать номер строки и столбца.

Количество индексов, используемых для доступа к элементам массива называется **размерность массива**.

Массивы.

Рассмотрим примеры двухмерных массивов:

```
// объявление двумерного массива
int[,] table1 = new int[3, 3];
// объявление трехмерного массива
int[,,] table2 = new int[2, 3 ,8];
// объявление и  инициализация двумерного массива
int[,] table3 = new int[3, 2] { {1, 10}, {15, 3}, {6, 3} };
```

Для доступа к элементу массива необходимо указать его индексы. Например:

```
table3[0,1]=4;
```

Массивы.

Ступенчатый **jagged** массив – это массив массивов. В нем длина каждого массива может быть разной.
Пример объявления ступенчатого массива:

```
static void Main(string[] args)
{
 // объявляем массив, который состоит из 3-х массивов
  int[][] array = new int[3][];
  array [0] = new int[3];          //создание подмассива
  array [1] = new int[2];          //создание подмассива
  array [2] = new int[5];          //создание подмассива
}
```

Доступ к элементам многомерных массивов:

```
array [0][1] = 55;
```

Класс System.Array

Все массивы в C# построены на основе базового класса **System.Array**, который содержит полезные для программиста свойства и методы.

Массивы. Свойства класса

Свойство **Length**

Т.к. массивы являются объектами у них есть свойства.

Свойство **Length**, возвращает количество элементов в массиве (во всех размерностях)

```
static void Main(string[] args)
{
   int[] vector = new int[7];
   int size = vector.Length;        // size = 7
  Console.WriteLine("size of vector = {0}", size);
}
```

Массивы. Свойства класса

Свойство **Length**

Когда запрашивается длина многомерного массива, то возвращается общее число элементов, из которых может состоять массив.

```
static void Main(string[] args)
{
   int[,] table3 = new int[3, 2] { {1, 10}, {15, 3}, {6, 3} };
  // size = 6
  Console.WriteLine("size of table3 = {0}", table3.Length);
}
```

Массивы. Свойства класса

В случае ступенчатого массива с помощью свойства **Length** можно получить длину каждого подмассива, составляющего ступенчатый массив. Рассмотрим такой пример:

```
int[][] myArr = new int[3][];
myArr[0] = new int[4];
myArr[1] = new int[10];
myArr[2] = new int[1];
Console.WriteLine(«Количество лент массива: " + myArr.Length);
Console.WriteLine("Длина первой ленты: " + myArr[0].Length);
Console.WriteLine("Длина второй ленты: " + myArr[1].Length);
Console.WriteLine("Длина третьей ленты: " + myArr[2].Length);
```

Массивы. Свойства класса

Двумерный ленточный массив представляет собой массив массивов.

Следовательно, когда используется выражение **myArr.Length** то в нем определяется число подмассивов, хранящихся в массиве **myArr** (в данном случае — 3 массива)

Массивы. Свойства класса

Свойство **Rank** показывает размерность массива

```
int [,] mas = new int [2,3];
Console.WriteLine(mas.Rank);
```

Результат: 2

Массивы. Методы класса

Метод **GetLength** возвращает длину заданного измерения массива:

```
int[,] table = new int[2, 2];
for (int i = 0; i < table.GetLength(0); i++)
{
    for (int j = 0; j < table.GetLength(1); j++)
     {
      table[i, j] = Convert.ToInt32(Console.ReadLine());
     }
}
```

Массивы. Методы класса

Метод **Clear()** позволяет очистить указанный диапазон элементов (числовые элементы приобретут значения 0, ссылки на объекты — **null**, логические элементы - **false**).
Первым параметром этого метода является имя массива, вторым – индекс, с которого происходит очистка, третьим – число элементов.

```
int [ ] c = {1, 2, 3, 4, 5};
Array.Clear(c,0,c.Length);
```

Массивы. Методы класса

Метод **GetLength()** используется для определения количества элементов в указанном измерении массива.

```
int [,] c = {{1,2,3},{4,5,6}};
int dim0 = c.GetLength(0);      // dim0 = 2
int dim1 = c.GetLength(1);      // dim1 = 3
```

Массивы. Методы класса

Метод **IndexOf()** возвращает номер первого вхождения указанного элемента. Если элемент не найден, то возвращается **-1**.

```
int [ ] c = {12,32,3,54,15,6};
int w = Array.IndexOf(c,54);        // w =3
```

Массивы. Методы класса

Метод **LastIndexOf()** возвращает номер последнего вхождения указанного элемента. Если элемент не найден, то возвращается -1.

```
int [ ] c = {12,32,3,54,12,6};
int w = Array.LastIndexOf(c,12);    // w =4
```

Массивы. Методы класса

Метод **Sort()** сортирует одномерный массив встроенных типов данных, причем массив передается как параметр.

```
int [ ] c = {12,32,3,54,15,6};
Array.Sort(c);
```

Результат :

3 6 12 15 32 54

Массивы. Методы класса

Метод **Reverse()** позволяет расставить элементы одномерного массива в обратном порядке, причем массив передается как параметр.

```
int [ ] c = {12,32,3,54,15,6};
Array.Reverse(c);
```

Результат :

6 15 54 3 32 12

Массивы. Методы класса

Метод **BinarySearch()** выполняет двоичный поиск в отсортированном массиве. Возвращает индекс элемента.

```
int[] c = { 12, 32, 3, 54, 15, 6 };
Array.Sort(c);
Console.WriteLine(Array.BinarySearch(c,
  12));
```

Массивы. **Копирование массивов**

Массивы — это ссылочные типы, поэтому присваивание переменной типа массива другой переменной создает две переменных, ссылающихся на один и тот же массив.

Для копирования массивов предусмотрена некоторые методы.

Метод **Clone()** создает неглубокую копию массива, т. е. если элементы массива относятся к типу значений, то все они копируются, если массив содержит элементы ссылочных типов, то сами эти элементы не копируются, а копируются лишь ссылки на них.

Массивы. **Копирование массивов**

В некоторых случаях вместо метода **Clone(),** можно также применять метод **Copy()**.

Но между **Clone()** и **Copy()** есть одно важное отличие: **Clone()** создает новый массив, а **Copy()** требует существующего массива той же размерности с достаточным количеством элементов.

Массивы. **Копирование массивов**

Метод **CopyTo()** используется для копирования элементов из исходного массива в массив назначения.

```
int[ ] c = { 12, 32, 3, 54, 15, 6 };
int[ ] d = new int[6];
c.CopyTo(d, 0);
```

Массивы. **Копирование массивов**

Метод **Copy()** используется для копирования заданного диапазона элементов из исходного массива в массив назначения.

```
int[] c = { 12, 32, 3, 54, 15, 6 };
int[] d = new int[6];
Array.Copy(c, d, 3);      // 12 32 3 0 0 0
Array.Copy(c,1, d, 2,3); // 0 0 32 3 54 0
```

Массивы.

Количество элементов массива фиксировано и задается при объявлении или инициализации массива.

Это не всегда удобно.

В C# имеется класс **List,** с помощью которого можно создавать динамический массив. Т.е. можно добавлять и удалять элементы в процессе работы программы.

Массивы.

Рассмотрим опять массив, который должен содержать цвета:

```
static void Main(string[] args)
{
  List<string> colours = new List<string>(); //
создание массива
  colours.Add("Barcelona"); // добавление элемента
  colours.Add("Chelsea");
  colours.Add("Arsenal");
 }
```

Массивы. Добавление элементов

Для добавления элементов в динамический массив, в C# реализовано несколько методов:

Метод	Описание
Add ([элемент])	добавляет элемент в конец списка
AddRange ([список элементов])	добавляет в конец списка элементы указанного списка
Insert ([индекс],[элемент])	вставляет элемент на позицию соответствующую индексу, все элементы «правее» будут сдвинуты на одну позицию вправо
InsertRange ([индекс], [список элементов])	то же самое, только вставляется несколько элементов

Массивы. Удаление элементов

Для удаления элементов можно использовать следующие методы:

Метод	Описание
Remove ([элемент])	удаляет первое вхождение указанного элемента из списка
RemoveAll ([элемент])	удаляет все вхождения указанного элемента из коллекции
RemoveRange ([индекс], [количество])	удаляет указанное количество элементов, начиная с указанной позиции
RemoveAt ([индекс])	удаляет элемент, который находится на указанной позиции
Clear()	удаляет все элементы списка

Массивы.

Для определения количества элементов можно использовать свойство **Count**:

Стоит отметить, что массивы работают быстрее, чем массивы заданные списками **List**.