Информационные технологии и программирование

Лекция 3. Основы языка C#

Содержание лекции:

- Типы вычислительных процессов
- Операторы выбора
- Массивы
- Циклы
- Стек и куча
- Особенности работы с массивами

Преподаватель курса: Клюкин Даниил Анатольевич, ст. преподаватель каф. ПМиИТ

Типы вычислительных процессов

Вычислительные процессы бывают 3 основных типов:

1. Линейный,
2. Разветвленные,
3. Цикличные.

В **линейном** процессе инструкции выполняются только последовательно.

В **разветвленных** программах предусматривается несколько вариантов инструкций в зависимости от выполненных условий.

Операторы выбора

1. Оператор **if**

```
if ([логическая конструкция])
{
   [Тело]
}
```

```
if ([логич. конструкция])
{
   [Тело 1]
}
else //Иначе
{
   [Тело 2]
}
```

```
var value = 5;
if (value > 0)
   Console.WriteLine("Положительное");

var value2 = 0;
if (value2 % 2 == 0)
   Console.WriteLine("Четное");
else
   Console.WriteLine("Нечётное");
```

```
if ([логич. Конструкция 1])
{
   [Тело 1]
}
else if ([логич. Конструкция 2])
{
   [Тело 2]
}
...
else if ([логич. Конструкция N])
{
   [Тело N]
}
else //Иначе
{
   [Тело N+1]
}
```

```
var value = 5;
if (value > 0)
   Console.WriteLine("Положительное");
else if (value < 0)
   Console.WriteLine("Отрицательное");
else
   Console.WriteLine("Равно 0");
```

2. Оператор **:?** (тернарный оператор)

```
int a=0;
int b=10;
if (b < 0)
{
   a = -1;
}
else
{
   a = 1;
}
```

эквивалентно

```
[тип данных] [имя переменной] = [логическая конструкция]
                              ? [значение если истина]
                              : [значение если ложь];
```

```
var b = 10;
var a = b < 0 ? -1 : 1;
```

3. Оператор **switch**

Конструкция switch/case оценивает некоторое выражение и сравнивает его значение с набором значений. И при совпадении значений выполняет определенный код.

```
switch ([значение переменной])
{
   case [шаблон 1]:
   {
      [Тело 1]
   } break;
   ...
   case [шаблон N]:
   {
      [Тело N]
   } break;
   default
   {
      [Тело N+1]
   } break;
}
```

```
switch (Console.ReadLine())
{
   case "0": Console.WriteLine(“Привет"); break;
   case "1": Console.WriteLine(“Пока"); break;
   default: Console.WriteLine("..."); break;
}
```

```
switch (int.Parse(Console.ReadLine()))
{
   case > 0: Console.WriteLine("+"); break;
   case < 0: Console.WriteLine("-"); break;
   default: Console.WriteLine("0"); break;
}
```

Блок `default` не обязателен.

Специальное слово `break` также не обязательно, оно говорит о выходе из текущего блока, если его нет, то будут проверяться остальные `case`.

Перечисления

Перечислимые константы (перечисления) – это тип значения, определенный набором именованных констант применяемого целочисленного типа.

Плохо: if (day == 1) … (Что такое 1? Понедельник? Вторник?)

Хорошо: if (day == DayOfWeek.Monday) … (Читается как английский текст).

Чтобы определить тип перечисления, используйте ключевое слово **enum** и укажите имена:

```
Transport myTransport = Transport.Car;

DayTime dayTime = DayTime.Evening;

// Создаем возможные транспортные средства
enum Transport { Car, Airplane, Bicycle, Boat, Spaceship }
// Создаем возможные времена дня
enum DayTime { Morning, Afternoon, Evening, Night}
```

```
enum DayTime { Morning, Afternoon, Evening, Night}
```

Перечисление возможно перевести в целое число, первое значение – **Morning** будет переведено в 0, **Afternoon** – в 1 и т.д.

```
DayTime dayTime = DayTime.Evening;

int intDayTime = (int)dayTime; // 2
```

Можно задать свои значения для каждой именованной константы:

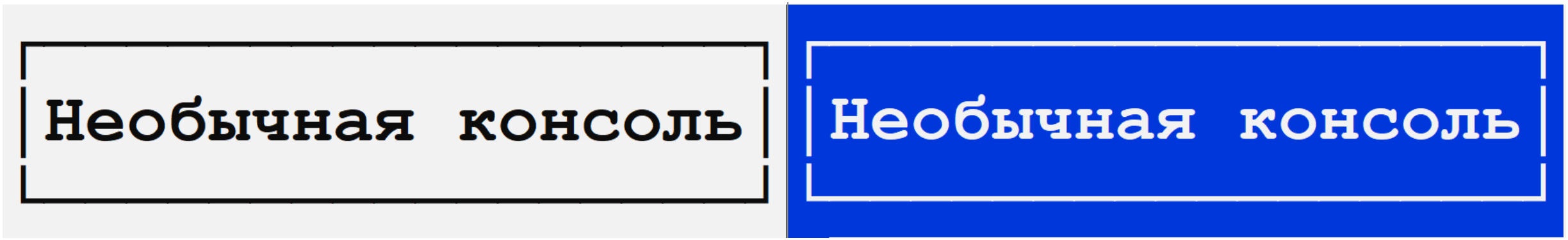
```
enum DayTime { Morning = 1, Afternoon = 2, Evening = 5, Night = 10 }
```

Перечисления используются для понятного обозначения возможных значений, например, изменим цвет нашей консоли:

```
Console.BackgroundColor = ConsoleColor.White;
Console.ForegroundColor = ConsoleColor.Black;

Console.WriteLine("                          ");
Console.WriteLine("  Необычная консоль       ");
Console.WriteLine("                          ");
```

Гораздо удобнее читать `ConsoleColor.White`, чем какое-нибудь «магическое число» 15.



Пример switch-expression:

```
var current = DayTime.Afternoon;

var message = current switch
{
    DayTime.Morning => "Доброе утро",
    DayTime.Afternoon => "Добрый день",
    DayTime.Evening => "Добрый вечер",
    DayTime.Night => "Доброй ночи",
    _ => "Неизвестное время суток"
};

Console.WriteLine(message);

enum DayTime { Morning, Afternoon, Evening, Night }
```

Вопросы

Что получится в результате операций:

```
int i = 0;

if (i < 10)
{
   i = 100;
}
if (i > 50)
{
   i *= 3;
}
else
{
   i *= 2;
}
```

```
int i = 0;

if (i < 10)
{
   i = 100;
}
else if (i > 50)
{
   i *= 3;
}
else
{
   i += 2;
}
```

```
int i = 5;
int j = 10;
int k = i < j ? i + j : i - j;
```

```
int a = 13;
int b = 5;
int c = a == b ? a : b;
```

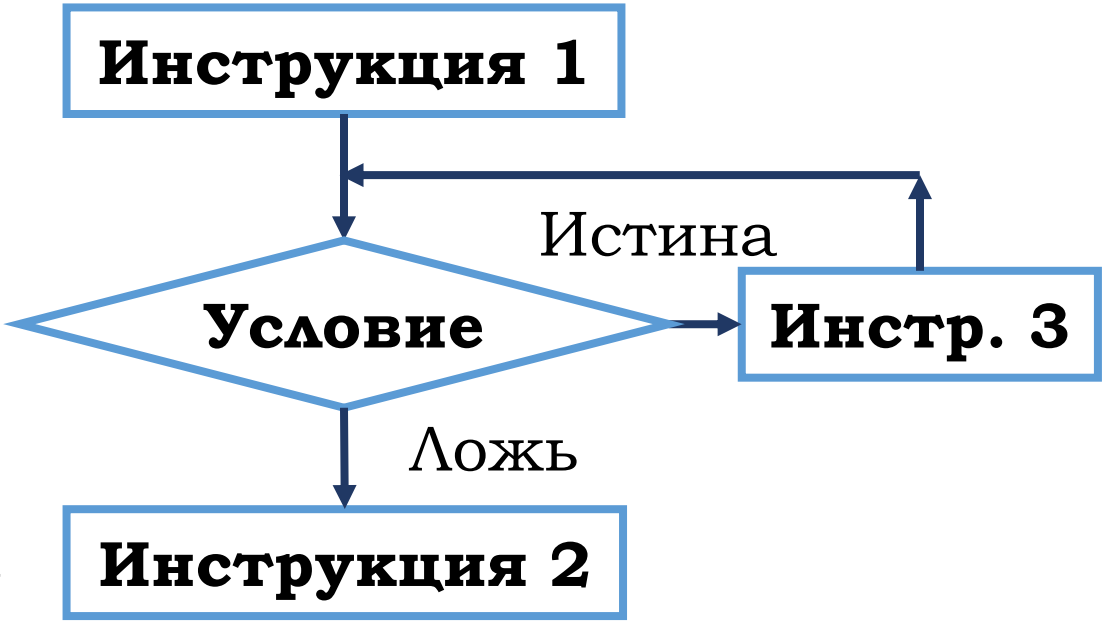
```
int q = 5;
int w = 1;
int e = q % 2 == w ? w : q;
```

```
var a = “Вася”;
var b = “Петя”;

switch (a)
{
   case “Коля”:
   {
      Console.WriteLine(1);
   } break;
   case “Петя”:
   {
      Console.WriteLine(2);
   } break;
   case “Вася”:
   {
      Console.WriteLine(3);
   } break;
}
```

Цикличные программы – когда действие необходимо выполнять множество раз пока выполняется какое-либо условие.

Виды циклов:



1. **Безусловные** (бесконечные) – когда условие выхода не предусмотрено
2. С **предусловием** – сначала проверяется условие, потом выполняется тело цикла.
3. С **постусловием** – сначала выполняется тело цикла, потом проверяется условие.
4. С **выходом из середины** – наиболее общая форма цикла. Синтаксически цикл оформляется с помощью: **начала** цикла, **конца** и **команды выхода**.
5. Совместный цикл (**цикл по коллекции**) – «Выполнить для всех элементов»

Массивы

Массив представляет собой набор однотипных данных. Объявление массива похоже на объявление переменной за тем исключением, что после указания типа ставятся квадратные скобки:

```
тип_переменных[] название_массива;
```

Инициализация массива:

```
название_массива = new тип_переменных[количество_элементов];
```

```
double[] array = new double[5];
```

```
int[] array2 = new int[10];
```

```
char[] array3 = new char[500];
```

Пример работы с массивом:

К элементам массива обращаются по **Индексу.**

Индексация начинается с **нуля**!

Создадим массив **длины 5**:

```
double[] array = new double[5]; Внутри: 0 0 0 0 0

array[0] = 3; Внутри: 3 0 0 0 0
array[2] = 7; Внутри: 3 0 7 0 0
```

В новых версиях C# возможно обращение с конца:

```
array[^1] = 5; Внутри: 3 0 7 0 5
array[^2] = 1; Внутри: 3 0 7 1 5
```

Чтобы узнать длину массива, необходимо вызвать свойство Length:

```
int arrayL = array.Length;
```

Пример работы с массивом:

Создадим и сразу заполним массив:

```
double[] array = new double[] {1, 3, 2, 5, 4};
```

Внутри: 1 3 2 5 4

Можно написать так:

```
double[] array = new [] {1.0, 3, 2, 5, 4};
```

Но тогда нужно явно указать, что числа вещественные написав знак после запятой хотя бы в одном числе: 1.0

Внутри: 1.0 3.0 2.0 5.0 4.0

С применением var:

```
var array = new [] {1.0, 3, 2, 5, 4};
```

Внутри: 1.0 3.0 2.0 5.0 4.0

Пример работы с массивом:

Создадим и сразу заполним массив:

```
double[] array = new double[] {1, 3, 2, 5, 4};
```

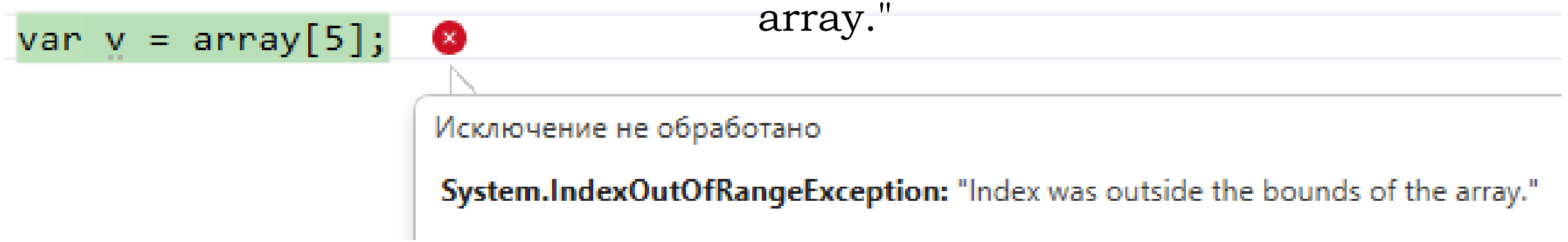
Индексы: 0 1 2 3 4

Внутри: 1 3 2 5 4

Обратимся к индексу 5.

```
var v = array[5];
```

System.IndexOutOfRangeException: "Index was outside the bounds of the array."



Циклы

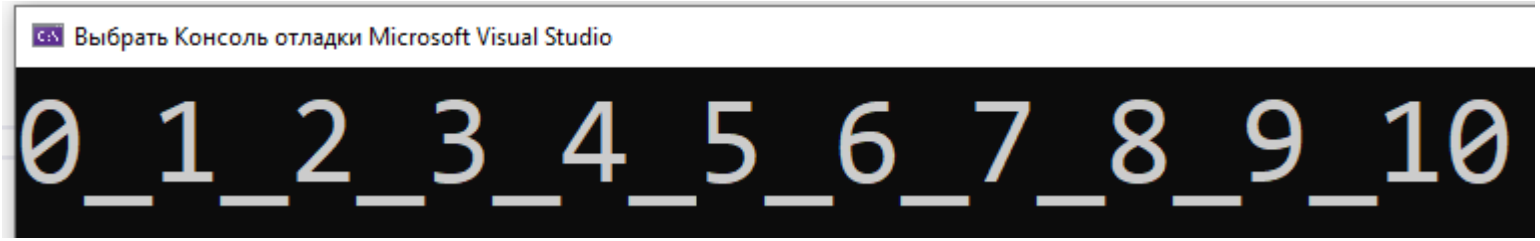
Итерация – повторение какого-либо действия.

1. Цикл for

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

Сначала проверяется условие, потом выполняется тело цикла, потом выполняется инкремент.

```
for (int i = 0; i <= 10; i++)
{
   Console.Write($"{i}" + (i == 10 ? "" : "_"));
}
```



2. Цикл do while

```
do
{
    действия цикла
}
while (условие)
```

```
int i = 0;

do
{
   i++;
}
while (i < 5);
```

Сначала выполняется тело цикла, потом проверяется условие.

3. Цикл while

```
while (условие)
{
    действия цикла
}
```

```
int i = 0;

while (i < 5)
{
   i++;
}
```

Сначала проверяется условие, потом выполняется тело цикла.

Пример

```
bool isPasswordCorrect;

do
{
    Console.WriteLine("Введите пароль:");

    var password = Console.ReadLine();

    isPasswordCorrect = password == "1234";

} while (isPasswordCorrect);
```

Вопросы

Мы хотим повторить действие **5 раз**, что должно стоять вместо **N**:

```
for (int i = 0; i <= N; i++)
{
   Console.Write($"{i}");
}

for (int i = 100; i > N; i--)
{
   Console.Write($"{i}");
}

for (int i = 0; i < N; i += 3)
{
   Console.Write($"{i}");
}
```

Специальное слово **continue** применяется для перехода к следующей итерации цикла

```
for (int i = 0; i < 1000; i++)
{
//Если число равно 100, то завершаем цикл
   if (i == 100)
       break;
//Если число четное, то переходим к следующей итерации
   if (i % 2 == 0)
       continue;
   Console.WriteLine(i);
}
```

Специальное слово **break** применяется для прерывания текущего блока кода, например, для остановки цикла.

Пример

```
while(true)
{
    Console.WriteLine("Введите пароль:");

    var password = Console.ReadLine();

    if (password == "1234")
        break;
}
```

Цикл foreach

Когда надо пробежаться по всем элементам коллекции

```
foreach (тип_данных переменная in коллекция)
{
    // действия цикла
}

var array = new double[] { 7, 1, 6, 9, 8 };

foreach (var t in array)
{
   Console.WriteLine(t);
}
```

Значимые и ссылочные типы

Пример с драматическим эффектом:

```
int[] a = { 1, 2, 3 };
int[] b = a; // Студент думает, что скопировал массив
b[0] = 99;
Console.WriteLine(a[0]); // Выведет 99!
```

Классификация типов по организации в памяти:

- **Значимые** типы: bool, byte, short, int, long, float, decimal, double, char, enum, struct.
- **Ссылочные** типы: string, массивы значений, классы, интерфейсы, object, делегаты.

Значимые типы — это типы, экземпляры которых хранят в памяти непосредственно своё значение.

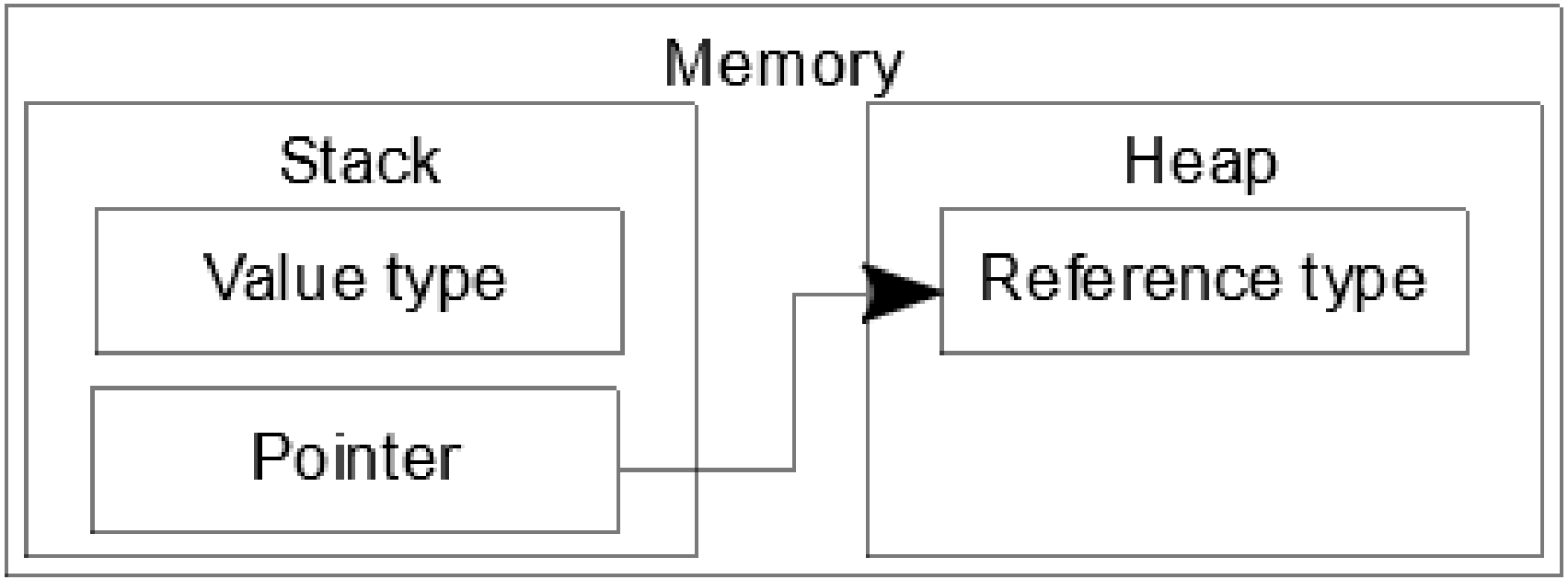
Ссылочные типы — это типы, экземпляры которых хранят адрес на область памяти где находятся данные.

Метафора:

Значимый тип (int): Это листок бумаги, на котором написано число. Когда мы делаем b = a, мы ксерокопируем листок. У каждого свой листок.

Ссылочный тип (Массив): Это **пульт от телевизора**. Сам телевизор (массив) стоит в Куче. Переменная a — это пульт. Когда мы делаем b = a, мы **копируем пульт**, а не телевизор. Теперь два пульта управляют одним телевизором.

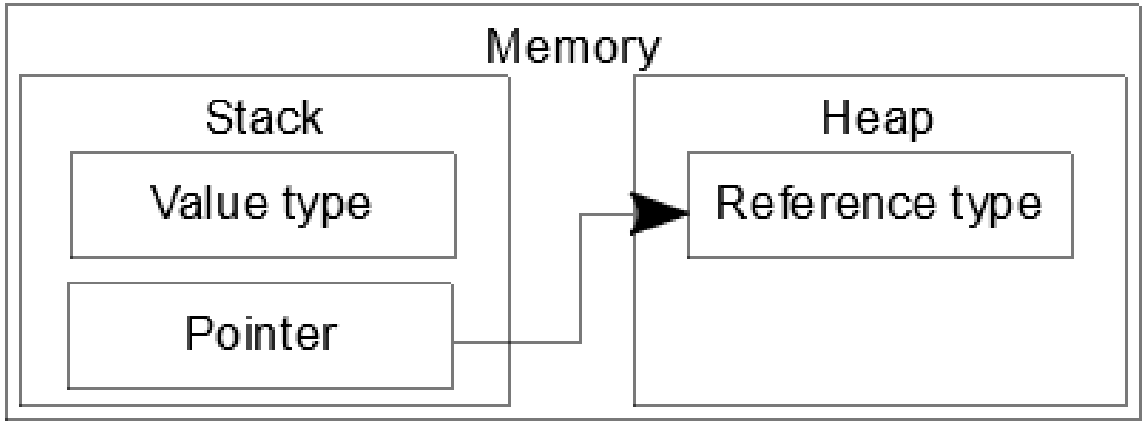
Стек и куча



Стек — это область памяти, организованная по принципу **LIFO** (Last In, First Out — «последним пришёл, первым ушёл»).

Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке.

Время жизни переменных таких типов ограничено их контекстом.



Куча — это динамически управляемая область памяти, предназначенная для хранения:

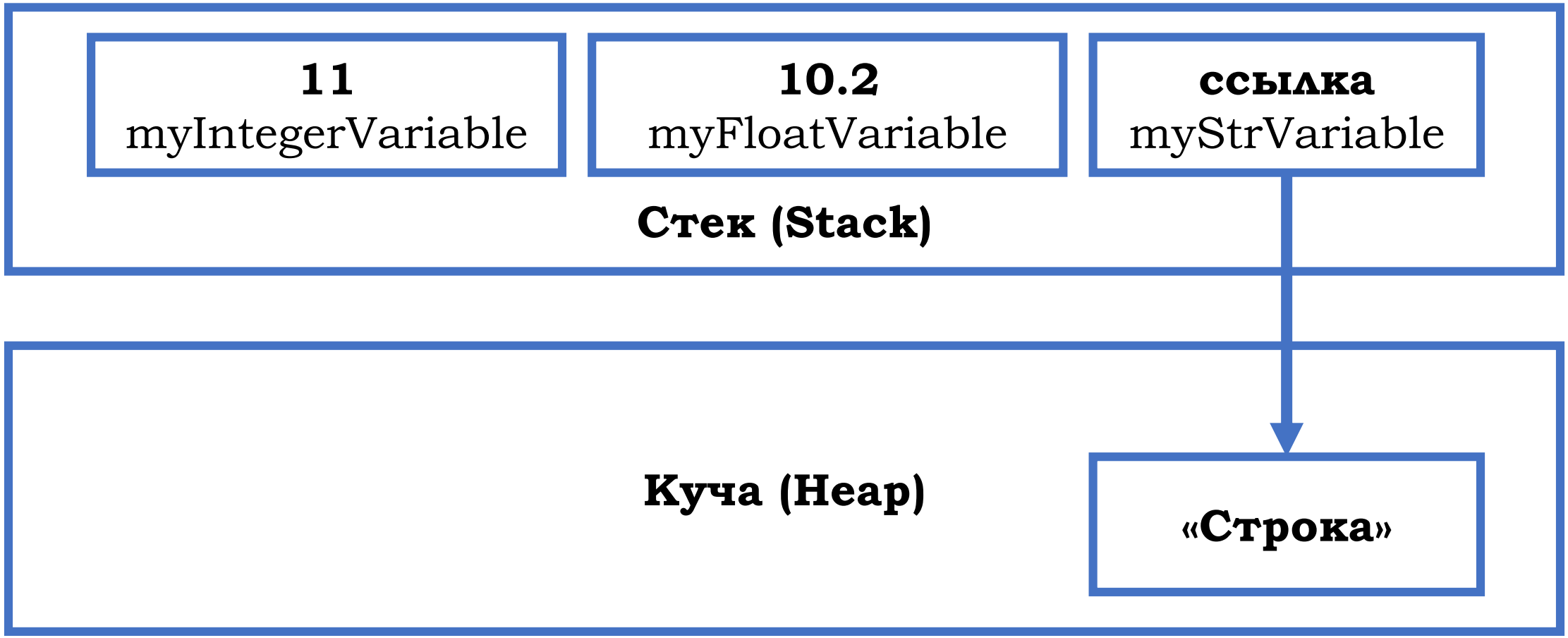
- объектов, созданных во время выполнения программы (например, через new в C# или C++),
- данных с неопределённым временем жизни (например, глобальные структуры, большие массивы).

Удаление данных из кучи происходит сборщиком мусора (Garbage Collector).

```
int myIntegerVariable = 11;

float myFloatVariable = 10.2f;

string myStrVariable = "Строка";
```

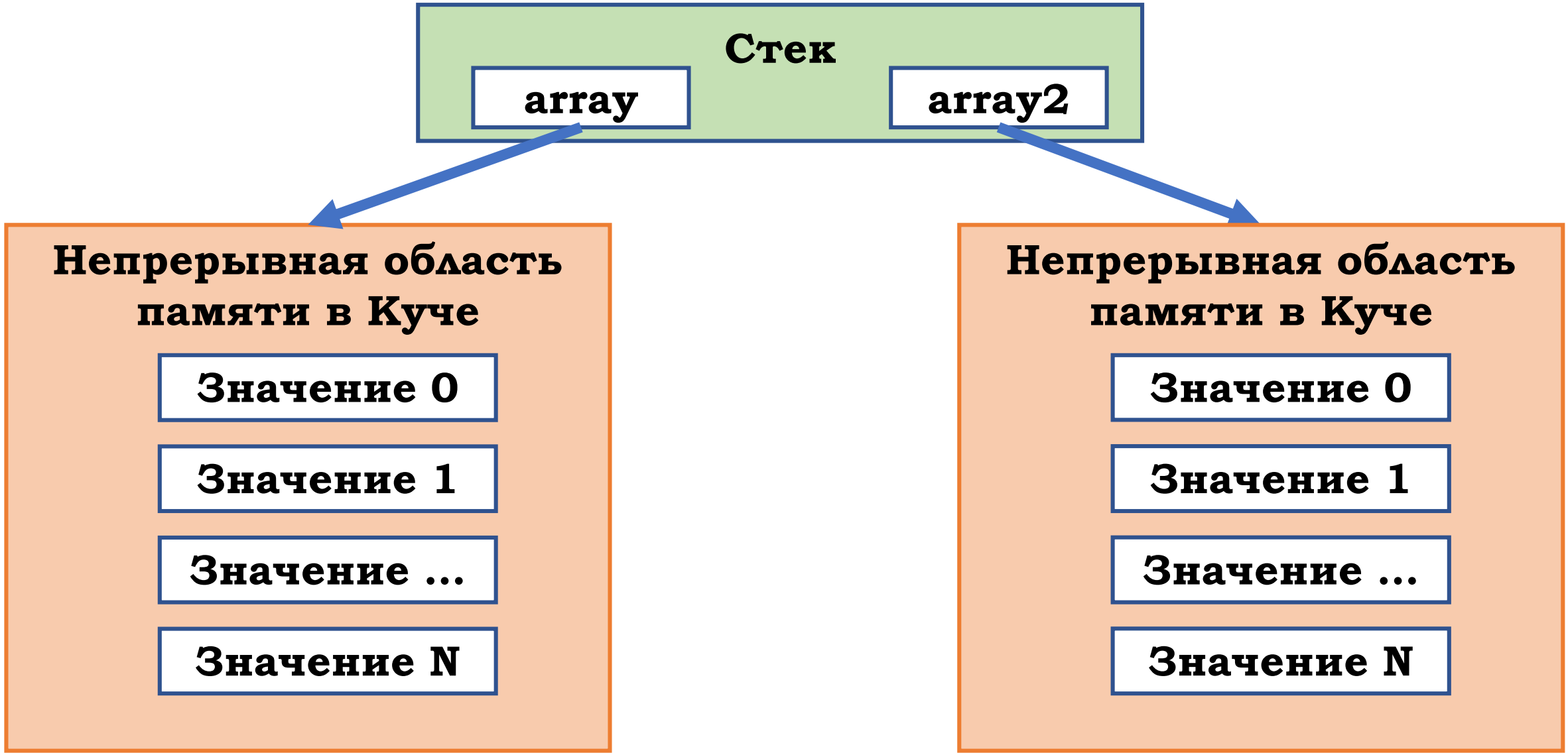


null — это специальное значение, указывающее на то, что переменная ссылочного типа (или nullable-типа) не ссылается ни на какой объект в памяти.

```
string text = null; // Допустимо (ссылочный тип)
```

Особенности работы с массивами

Создание 2 массивов выглядит следующим образом в памяти:

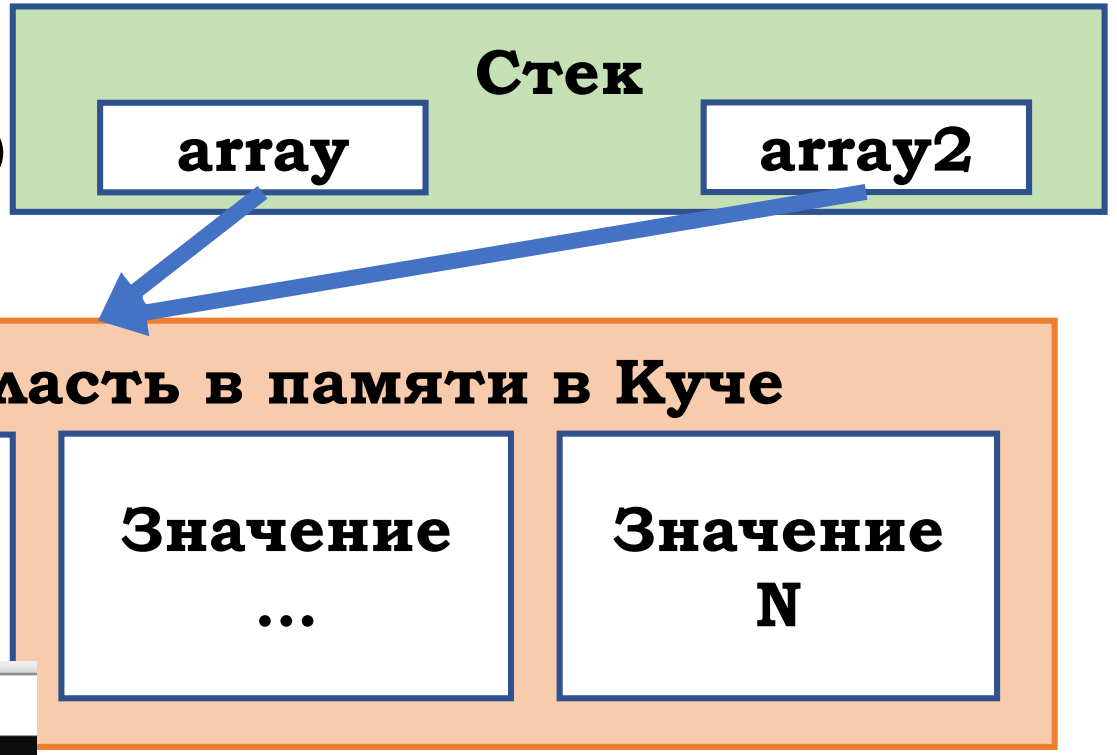


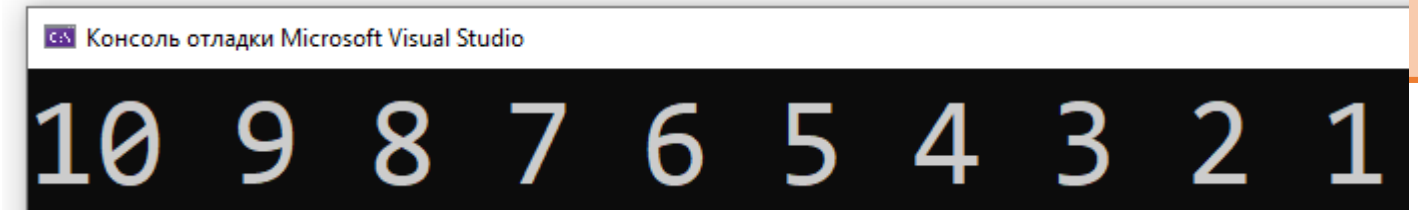
```
double[] array = new double[10];
double[] array2 = new double[10];

array2 = array; // Теперь ссылка array2 указывает на ту же область памяти, что и у array
for (int i = 0; i < array.Length; i++)
    array[i] = array.Length - i;

for (int i = 0; i < array.Length; i++)
    Console.Write($"{array2[i]} ");
```

ВАЖНО!
Массивы – ссылочные типы данных.





Диапазон

В **C# 8.0** была добавлена новая функциональность - индексы и диапазоны, которые упрощают получение из массивов подмассивов. Для этого в **C#** есть два типа: **System.Range** и **System.Index**. Оба типа являются структурами. Тип **Range** представляет некоторый диапазон значений в некоторой последовательность, а тип **Index** - индекс в последовательности.

```
Range range = 3..7;   // с 3 индекса до 6
```

Диапазон представляет часть последовательности, которая ограничена двумя индексами. Начальный индекс включается в диапазон, а конечный индекс НЕ входит в диапазон. Для определения диапазона применяется оператор «**..**»:

```
Range range = 3..7;   // с 3 индекса до 6
Range range2 = 2..;   // с 2 индекса до конца
Range range3 = 1..^1; // с 1 индекса до предпоследнего
var arr1 = arr[range];  // 3, 4, 5, 6
var arr2 = arr[range2]; // 2, 3, 4, 5, 6, 7, 8, 9
var arr3 = arr[range3]; // 1, 2, 3, 4, 5, 6, 7, 8
```

Многомерный массив

Массивы характеризуются таким понятием как **ранг** или количество измерений. **Выше** мы рассматривали массивы, которые имеют одно измерение (то есть **их ранг равен 1**) - такие массивы можно представлять в виде ряда (строки или столбца) элемента.
Но массивы также бывают **многомерными**. У таких массивов количество измерений (то есть ранг) **больше 1**.
Массивы которые имеют два измерения (ранг равен 2) называют **двухмерными**.

```
тип_переменной[,] название_массива
 = new тип_переменной[кол-во_строк, кол-во_столбцов];

тип_переменной[,,] название_массива = new тип_переменной[N, M, K];

И т.д.
```

Пример

Создадим массив с 2 строками и 3 столбцами:

```
double[,] array = new double[2,3];
```

Внутри массива:

0 0 0

0 0 0

```
array[0,0] = 1;
array[1,2] = 2;
```

Внутри массива:

1 0 0

0 0 2

```
array[0,1] = 3;
array[1,0] = 4;
```

Внутри массива:

1 3 0

4 0 2

Пример 2

Создадим массив с 3 строками и 4 столбцами и сразу заполним его:

```
var array = new [,]
{
   {1, 0, 2, 3},
   {2, 6, 5, 0},
   {9, 8, 4, 2}
};
```

Внутри массива:

```
1 0 2 3
2 6 5 0
9 8 4 2
```

Чтобы узнать общую длину массива (общее количество элементов):

```
array.Length;
```

Чтобы узнать длину массива по заданной размерности:

Количество строк:

```
array.GetLength(0);
```

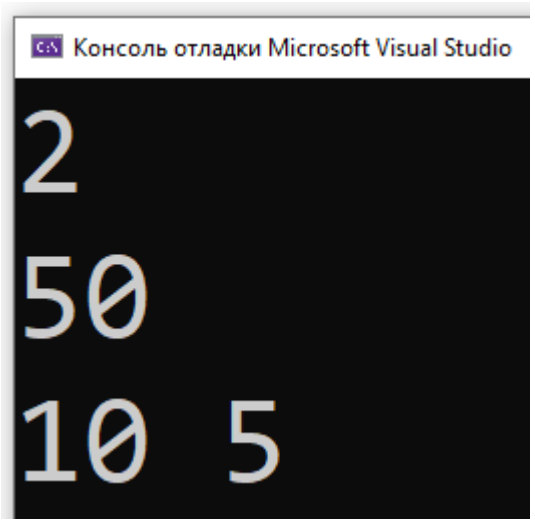
Количество столбцов:

```
array.GetLength(1);
```

```
double[,] array = new double [10, 5];

Console.WriteLine(array.Rank);
Console.WriteLine(array.Length);
Console.WriteLine($"{array.GetLength(0)} {array.GetLength(1)}");

for (int i = 0; i < array.GetLength(0); i++)
{
   for (int j = 0; j < array.GetLength(1); j++)
   {
      array[i, j] = i * array.GetLength(1) + j;
   }
}
```



Массивы массивов.

```
double[][] array = new double[10][];

Console.WriteLine(array.Rank);
Console.WriteLine(array.Length);

for (int i = 0; i < array.Length; i++)
{
   array[i] = new double[i + 1];

   for (int j = 0; j < array[i].Length; j++)
   {
      array[i][j] = j * array[i].Length/2 + i;

      Console.Write($"{array[i][j]} ");
   }
   Console.WriteLine();
}
```