№ 2 Основы CLRи .NET.Типы. Массивы, кортежи и строки

**Курносенко Софья**

**ТИПЫ**

**1 (a) Определите переменные всех возможных примитивных типов С# и проинициализируйте их. Осуществите ввод и вывод их значений, используя консоль.**

Как и во многих языках программирования, в C# есть своя система типов данных, которая используется для создания переменных. Тип данных определяет внутреннее представление данных, множество значений, которые может принимать объект, а также допустимые действия, которые можно применять над объектом.

В языке C# есть следующие примитивные (значимые) типы данных:

* **bool**: хранит значение true или false (логические литералы). Представлен системным типом System.Boolean

|  |  |
| --- | --- |
| 1  2 | bool alive = true;  bool isDead = false; |

* **byte**: хранит целое число от 0 до 255 и занимает 1 байт. Представлен системным типом System.Byte

|  |  |
| --- | --- |
| 1  2 | byte bit1 = 1;  byte bit2 = 102; |

* **sbyte**: хранит целое число от -128 до 127 и занимает 1 байт. Представлен системным типом System.SByte

|  |  |
| --- | --- |
| 1  2 | sbyte bit1 = -101;  sbyte bit2 = 102; |

* **short**: хранит целое число от -32768 до 32767 и занимает 2 байта. Представлен системным типом System.Int16

|  |  |
| --- | --- |
| 1  2 | short n1 = 1;  short n2 = 102; |

* **ushort**: хранит целое число от 0 до 65535 и занимает 2 байта. Представлен системным типом System.UInt16

|  |  |
| --- | --- |
| 1  2 | ushort n1 = 1;  ushort n2 = 102; |

* **int**: хранит целое число от -2147483648 до 2147483647 и занимает 4 байта. Представлен системным типом System.Int32. Все целочисленные литералы по умолчанию представляют значения типа int:

|  |  |
| --- | --- |
| 1  2  3 | int a = 10;  int b = 0b101;  // бинарная форма b =5  int c = 0xFF;   // шестнадцатеричная форма c = 255 |

* **uint**: хранит целое число от 0 до 4294967295 и занимает 4 байта. Представлен системным типом System.UInt32

|  |  |
| --- | --- |
| 1  2  3 | uint a = 10;  uint b = 0b101;  uint c = 0xFF; |

* **long**: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт. Представлен системным типом System.Int64

|  |  |
| --- | --- |
| 1  2  3 | long a = -10;  long b = 0b101;  long c = 0xFF; |

* **ulong**: хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт. Представлен системным типом System.UInt64

|  |  |
| --- | --- |
| 1  2  3 | ulong a = 10;  ulong b = 0b101;  ulong c = 0xFF; |

* **float**: хранит число с плавающей точкой от -3.4\*1038 до 3.4\*1038 и занимает 4 байта. Представлен системным типом System.Single

Все вещественные литералы рассматриваются как значения типа *double*. И чтобы указать, что дробное число представляет тип float или тип decimal, необходимо к литералу добавлять суффикс

* Литерал с суффиксом f или F относится к типу float
* Литерал с суффиксом m или M относится к типу decimal
* Литерал без суффикса или с суффиксом d или D относится к типу double

double d = 3D;

d = 4d;

d = 3.934\_001;

float f = 3\_000.5F;

f = 5.4f;

decimal myMoney = 3\_000.5m;

myMoney = 400.75M;

* **double**: хранит число с плавающей точкой от ±5.0\*10-324 до ±1.7\*10308 и занимает 8 байта. Представлен системным типом System.Double
* **decimal**: хранит десятичное дробное число. Если употребляется без десятичной запятой, имеет значение от ±1.0\*10-28 до ±7.9228\*1028, может хранить 28 знаков после запятой и занимает 16 байт. Представлен системным типом System.Decimal
* **char**: хранит одиночный символ в кодировке Unicode и занимает 2 байта. Представлен системным типом System.Char. Этому типу соответствуют символьные литералы:

|  |  |
| --- | --- |
| 1  2  3 | char a = 'A';  char b = '\x5A';  char c = '\u0420'; |

|  |  |  |
| --- | --- | --- |
| Тип C# | Размер  вбитах | Тип System(FCL) |
| sbyte | 8 | System.Sbyte |
| short | 16 | System.Int16 |
| int | 32 | System.Int32 |
| long | 64 | System.Int64 |
| byte | 8 | System.Byte |
| ushort | 16 | System.Uint16 |
| uint | 32 | System.Uint32 |
| ulong | 64 | System.Uint64 |
| char | 16 | System.Char |
| bool | 8 | System.Boolean |
| float | 32 | System.Single |
| double | 64 | System.Double |
| decimal | 128 | System.Decimal |
| string | - | System.String |
| object | - | System.Object |

**КОД:**

//хранит значение true или false (логические литералы)

bool f = false;

bool t = true;

//хранит целое число от 0 до 255 и занимает 1 байт

byte byte\_num = 255;

//хранит целое число от -128 до 127 и занимает 1 байт

sbyte sbyte\_neg\_num = 10;

sbyte sbyte\_pos\_num = -65;

// хранит целое число от -32768 до 32767 и занимает 2 байта

short sh\_neg\_num = -4000;

short sh\_pos\_num = 2768;

//хранит целое число от 0 до 65535 и занимает 2 байта

ushort ush\_num = 64590;

//хранит целое число от -2147483648 до 2147483647 и занимает 4 байта

//Все целочисленные литералы по умолчанию представляют значения типа int

int i\_hex\_num = 0xFF; //i\_hex\_num = 255

int i\_dec\_num = -999;

//хранит целое число от 0 до 4294967295 и занимает 4 байта

uint ui\_num = 80;

//хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт

long l\_neg\_num = -602;

long l\_pos\_num = 5339;

//хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт

ulong ul\_num = 0x3A; //ul\_num = 58

//хранит число с плавающей точкой от ± 1,5\*10^-45 до ± 3,4\*10^38, точность ~15-17 цифр и занимает 4 байта

//В visual studio дробное значение по умолчанию воспринимается как тип double, чтобы явно указать,

//что у нас число типа float можно использовать в конце суффикс F/f

float f\_neg\_num = -214.3f;

float f\_pos\_num = 67.19f;

//хранит число с плавающей точкой от ±5.0\*10^-324 до ±1.7\*10^308, точность ~6-9 цифр и занимает 8 байт

double d\_neg\_num = -256.16e34;

double d\_pos\_num = 45;

//хранит число с плавающей точкой от ±1.0\*10^-28 до ±7.9228\*10^28, точность 28-29 цифр и занимает 16 байт

decimal dec\_neg\_num = -34.92223m;

decimal dec\_pos\_num = 7.21m;

//хранит одиночный символ в кодировке Unicode и занимает 2 байта

char symb = 'S';

Console.WriteLine($"Bool : {f}, {t} \n" +

$"Byte : {byte\_num} \n" +

$"Sbyte : {sbyte\_neg\_num}, {sbyte\_pos\_num} \n" +

$"Short : {sh\_neg\_num}, {sh\_pos\_num} \n" +

$"Ushort : {ush\_num} \n" +

$"Int : {i\_hex\_num}, {i\_dec\_num} \n" +

$"Uint : {ui\_num} \n" +

$"Long : {l\_neg\_num}, {l\_pos\_num} \n" +

$"Ulong : {ul\_num} \n" +

$"Float : {f\_neg\_num}, {f\_pos\_num} \n" +

$"Double : {d\_neg\_num}, {d\_pos\_num} \n" +

$"Decimal : {dec\_neg\_num}, {dec\_pos\_num} \n" +

$"Char : {symb} \n");

**2(b) Выполните 5 операций явного и 5 неявного приведения. Изучите возможности класса *Convert*.**

Иногда может потребоваться скопировать значение в переменную или параметр метода другого типа. Такого рода операции называются преобразованиями типа. Разберем неявные и явные преобразования.

**► Неявное преобразование (автоматическое)**. Осуществляется при назначении переменной, занимающей большее количество байт в памяти, меньшей переменной.

Например, переменная типа [long](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) (64-разрядное целое число) может хранить любое значение, которое может хранить переменная [int](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) (32-разрядное целое число).

В следующей таблице приведены предопределенные неявные преобразования между встроенными числовыми типами:

| **НЕЯВНЫЕ ЧИСЛОВЫЕ ПРЕОБРАЗОВАНИЯ** | |
| --- | --- |
| **Исходный тип** | **Во что преобразовывается** |
| [sbyte](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | short, int, long, float, double, decimal или nint |
| [byte](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | short, ushort, int, uint, long, ulong, float, double, decimal, nint или nuint |
| [short](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | int, long, float, double или decimal либо nint |
| [ushort](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | int, uint, long, ulong, float, double или decimal, nint или nuint |
| [int](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | long, float, double или decimal, nint |
| [uint](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | long, ulong, float, double или decimal либо nuint |
| [long](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | float, double или decimal |
| [ulong](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | float, double или decimal |
| [float](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | double |
| [nint](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/nint-nuint) | long, float, double или decimal |
| [nuint](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/nint-nuint) | ulong, float, double или decimal |

**Примечание**

Неявные преобразования из int, uint, long, ulong, nint или nuint в float и из long, ulong, nint или nuint в double могут приводить к потере точности, но не к потере порядка величин. Другие неявные числовые преобразования никогда не теряют никаких сведений.

Также обратите внимание на следующее.

* Любой [целочисленный тип](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) неявно преобразуется в любой [числовой тип с плавающей запятой](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types).
* Не поддерживается неявное преобразование в типы byte и sbyte. Не поддерживается неявное преобразование из типов double и decimal.
* Не поддерживается неявное преобразование между типом decimal и типами float или double.

Также можно преобразовать значение типа **char** в числовое, занимающее большее количество байт, например, в int, float и т.д.

**►** Тем не менее если преобразование нельзя выполнить без риска потери данных, компилятор требует выполнения **явного преобразования**, которое называется приведением. **Приведение** — это способ явно указать компилятору, что необходимо выполнить преобразование и что вам известно, что может произойти потеря данных или приведение может завершиться сбоем во время выполнения. Чтобы выполнить приведение, укажите тип, в который производится приведение, в круглых скобках перед преобразуемым значением или переменной.

В следующей таблице показаны предопределенные явные преобразования между встроенными числовыми типами, для которых нет [неявного преобразования](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/numeric-conversions#implicit-numeric-conversions):

| **ЯВНЫЕ ЧИСЛОВЫЕ ПРЕОБРАЗОВАНИЯ** | |
| --- | --- |
| **Исходный тип** | **Во что преобразовывается** |
| [sbyte](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | byte, ushort, uint или ulong либо nuint |
| [byte](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte |
| [short](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, ushort, uint, ulong или nuint |
| [ushort](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byteили short |
| [int](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort, uint, ulong или nuint |
| [uint](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort или int |
| [long](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort, int, uint, ulong, nint или nuint |
| [ulong](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types) | sbyte, byte, short, ushort, int, uint, long, nint или nuint |
| [float](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | sbyte, byte, short, ushort, int, uint, long, ulong, decimal, nint или nuint |
| [double](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | sbyte, byte, short, ushort, int, uint, long, ulong, float, decimal, nint или nuint |
| [decimal](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types) | sbyte, byte, short, ushort, int, uint, long, ulong, float, double, nint или nuint |
| [nint](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/nint-nuint) | sbyte, byte, short, ushort, int, uint, ulong или nuint |
| [nuint](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/nint-nuint) | sbyte, byte, short, ushort, int, uint, long или nint |

Попробуем применить сложение к двум объектам типа **byte**:

byte a = 4;

byte b = a + 70;

При операциях мы должны учитывать диапазон значений, которые может хранить тот или иной тип. Но в данном случае число 74, которое мы ожидаем получить, вполне укладывается в диапазон значений типа byte, тем не менее, мы получаем ошибку.

Дело в том, что операция сложения (да и вычитания) возвращает значение типа int, если в операции участвуют целочисленные типы данных с разрядностью меньше или равно int (то есть типы byte, short, int). Поэтому результатом операции a + 70 будет объект, который имеет длину в памяти 4 байта. Затем этот объект мы пытаемся присвоить переменной b, которая имеет тип byte и в памяти занимает 1 байт.

И чтобы выйти из этой ситуации, необходимо применить операцию преобразования типов:

byte a = 20;

byte b = (byte)(a + 2);

Операция преобразования типов предполагает указание в скобках того типа, к которому надо преобразовать значение.

**КОД:**

//неявное преобразование

byte byte\_a = 30;

int int\_a = byte\_a;

char symb = 'A';

float int\_symb = symb;

byte byte\_b = 10;

float float\_b = byte\_b;

long long\_c = 9872;

double double\_c = long\_c;

short short\_d = -45;

int int\_d = short\_d;

Console.WriteLine($"Неявное преобразование \n" +

$"byte -> int : {byte\_a} \n" +

$"char -> float : {int\_symb} \n" +

$"byte -> float : {float\_b} \n" +

$"long -> double : {double\_c} \n" +

$"short -> int : {int\_d}");

//явное преобразование

sbyte sbyte\_e = -1;

byte byte\_e = (byte)sbyte\_e;

float float\_f = 9.23f;

byte byte\_f = (byte)float\_f;

double double\_g = 12.9887615213456;

float float\_g = (float)double\_g;

int int\_j = -289;

sbyte sbyte\_j = (sbyte)int\_j;

int int\_k = 10;

short short\_k = (short)int\_k;

Console.WriteLine($"\nЯвное преобразование \n" +

$"sbyte -> byte : {byte\_e} \n" +

$"float -> byte : {byte\_f} \n" +

$"double -> float : {float\_g} \n" +

$"int -> byte : {sbyte\_j} \n" +

$"int -> short : {short\_k}");

► **Преобразования с использованием вспомогательных классов**. Чтобы выполнить преобразование между несовместимыми типами, например целыми числами и объектами [System.DateTime](https://docs.microsoft.com/ru-ru/dotnet/api/system.datetime) или шестнадцатеричными строками и массивами байтов, можно использовать классы [System.BitConverter](https://docs.microsoft.com/ru-ru/dotnet/api/system.bitconverter) и [System.Convert](https://docs.microsoft.com/ru-ru/dotnet/api/system.convert), а также методы Parse встроенных числовых типов, такие как [Int32.Parse](https://docs.microsoft.com/ru-ru/dotnet/api/system.int32.parse).

**Класс Convert** преобразует значение одного базового типа к другому.

Для этого в нем определены следующие статические методы:

* **ToBoolean(value)**
* **ToByte(value)**
* **ToChar(value)**
* **ToDateTime(value)**
* **ToDecimal(value)**
* **ToDouble(value)**
* **ToInt16(value)**
* **ToInt32(value)**
* **ToInt64(value)**
* **ToSByte(value)**
* **ToSingle(value)**
* **ToUInt16(value)**
* **ToUInt32(value)**
* **ToUInt64(value)**

Метод Console.ReadLine() считываетданные с консоли и возвращает их как строку. Но, если нам необходимо считать значение другого типа, нужно использовать методы класса Convert для явного преобразования:

**КОД:**

Console.Write("Enter your name : ");

string name = Console.ReadLine();

Console.Write("Age : ");

int age = Convert.ToByte(Console.ReadLine());

Console.WriteLine($"Name : {name}, age : {age}");

**2(c) Выполните упаковку и распаковку значимых типов.**

Ранее мы рассматривали следующие элементарные типы данных: int, byte, double, string, object и др. Также есть сложные типы: структуры, перечисления, классы. Все эти типы данных можно разделить на типы значений, еще называемые **значимыми типами**, (value types) и ссылочные типы (reference types). Важно понимать между ними различия.

Типы значений:

* Целочисленные типы (byte, sbyte, short, ushort, int, uint, long, ulong)
* Типы с плавающей запятой (float, double)
* Тип decimal
* Тип bool
* Тип char
* Перечисления enum
* Структуры (struct)

Ссылочные типы:

* Тип object
* Тип string
* Классы (class)
* Интерфейсы (interface)
* Делегаты (delegate)

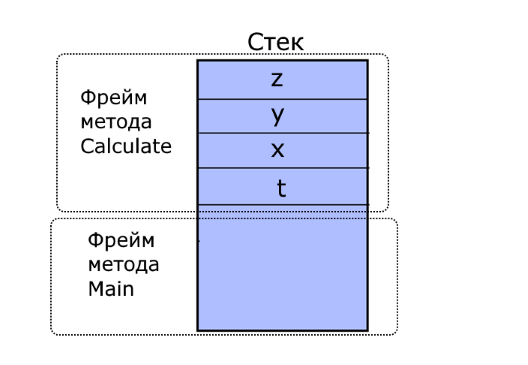
В чем же между ними различия? Для этого надо понять организацию памяти в .NET. Здесь память делится на два типа: **стек** и **управляемая куча** (heap). Значимые типы хранятся в стеке, а ссылочные в куче. Физически стек – это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека, устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class Program  {      static void Main(string[] args)      {          Calculate(5);          Console.ReadKey();      }        static void Calculate(int t)      {          int x = 6;          int y = 7;          int z = y + t;      }  } |

При запуске такой программы в стеке будут определяться два фрейма - для метода Main (так как он вызывается при запуске программы) и для метода Calculate:



При вызове этого метода Calculate в его фрейм в стеке будут помещаться значения t, x, y и z. Они определяются в контексте данного метода. Когда метод отработает, область памяти, которая выделялась под стек, впоследствии может быть использована другими методами.

Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредсвенное значение этого параметра или переменной. Например, в данном случае переменные и параметр метода Calculate представляют значимый тип - тип int, поэтому в стеке будут храниться их числовые значения.

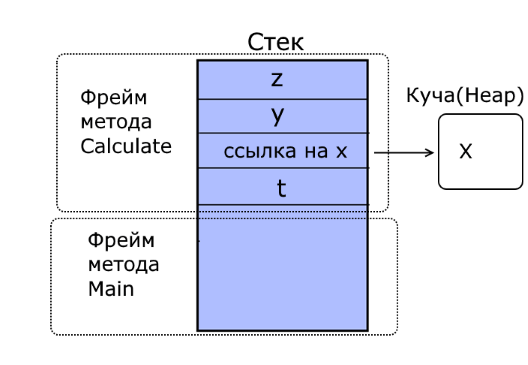
**Ссылочные типы хранятся в куче или хипе**, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

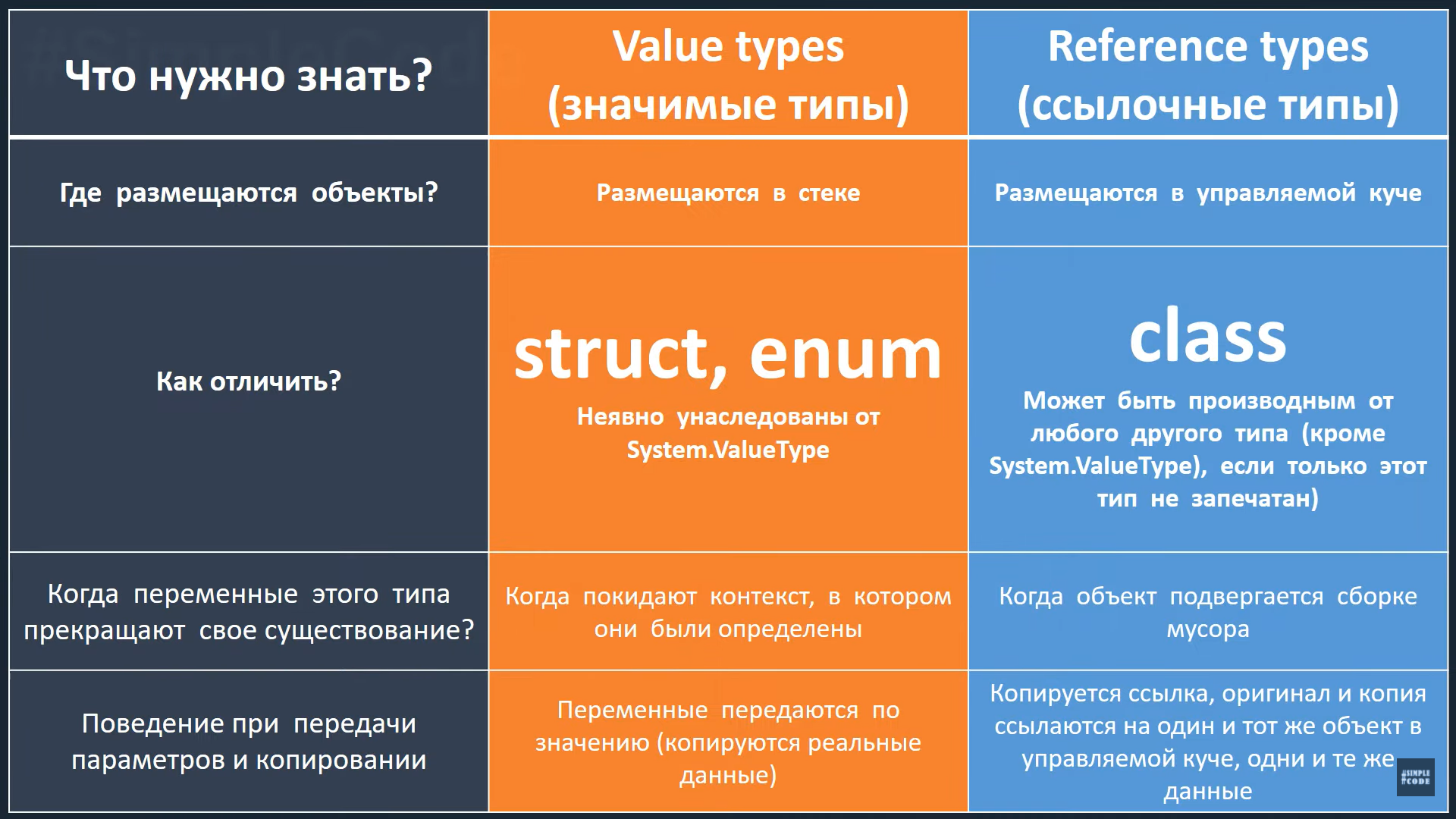
При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе больше нет ссылок, условно удаляет этот объект и очищает память – фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.

Так, в частности, если мы изменим метод Calculate следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | static void Calculate(int t)  {      object x = 6;      int y = 7;      int z = y + t;  } |

То теперь значение переменной x будет храниться в куче, так как она представляет ссылочный тип object, а в стеке будет храниться ссылка на объект в куче.





**Упаковка и распаковка значимых типов**

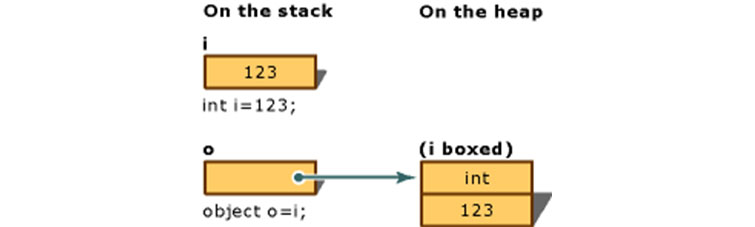
*Упаковка (boxing)* – размещение значимого типа в куче, *распаковка (unboxing)* – извлечение значимого типа из кучи.

* Когда любой значимый тип присваивается к ссылочному типу данных, значение перемещается из области стека в кучу. Эта операция называется **упаковкой**.
* Когда любой ссылочный тип присваивается к значимому типу данных, значение перемещается из области кучи в стек. Это называется **распаковкой**.

К примеру, здесь мы имеем следующий пример упаковки:



А вот состояние памяти в момент произведения операции:

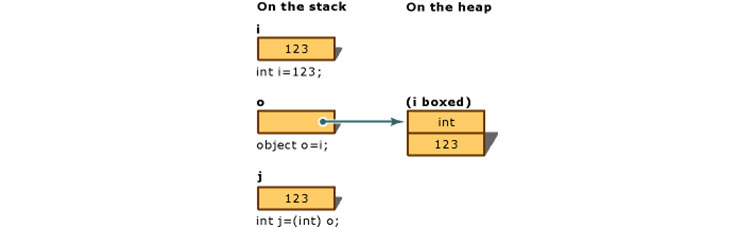


Чтобы сохранить значение "123" в виде объекта, в куче создается "упаковка", куда впоследствии и перемещаются данные.

Когда же производится распаковка:



Вот что происходит с памятью:

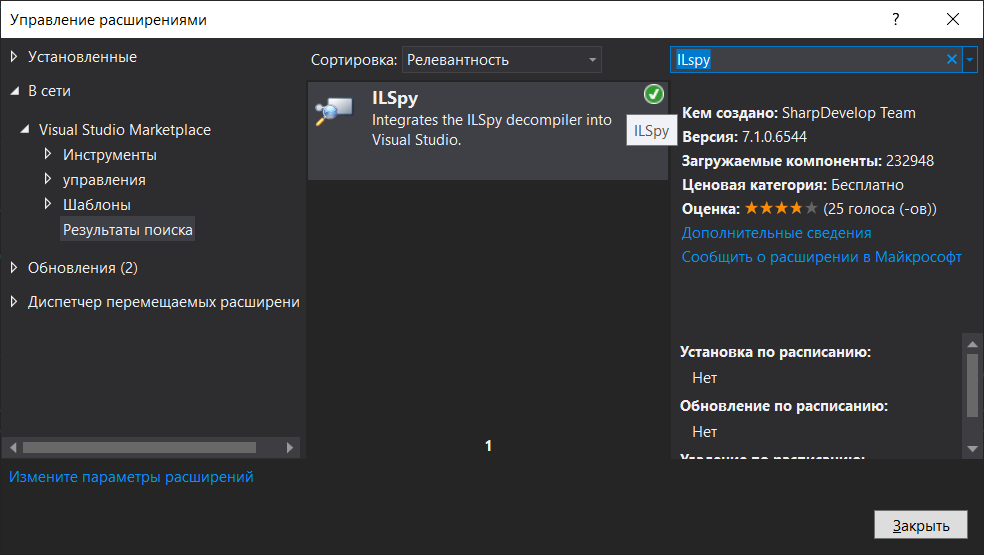


Значение "123" было изъято из упаковки и помещено назад в область стека.

Заметьте, что когда тип данных i упаковывается внутри объекта o, в стеке хранится лишь ссылка, в то время как само значение хранится в куче. Как только производиться распаковка, данные в куче обязаны быть скопированы в стек (переменная j). В обоих случаях наша цель - это работать с тем самым значением (123).

Расширения –> Управление расширениями

Установила ILspy, теперь можно просмотреть, где в коде были операции упаковки/распаковки.



ILSpy - это обозреватель и декомпилятор сборки .NET с открытым исходным кодом.

**КОД:**

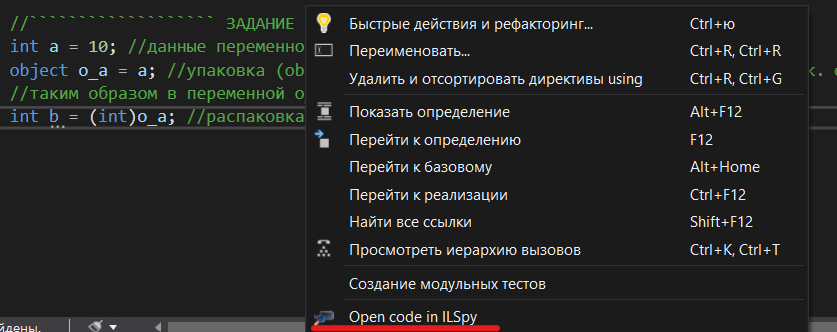
**//явная упаковка/распаковка**

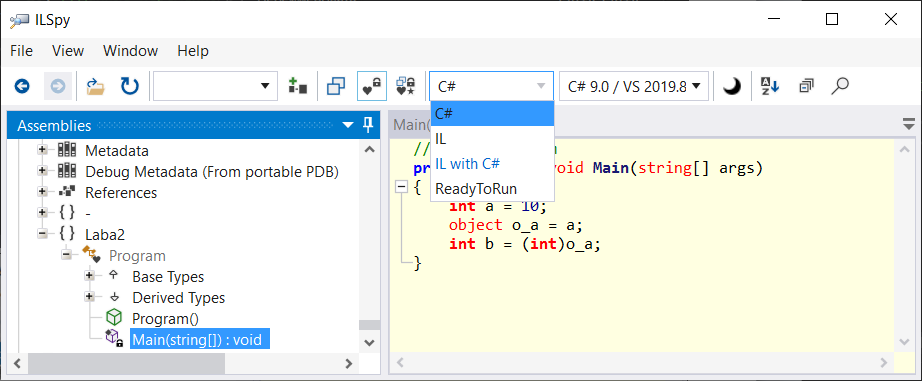
int a = 10; //данные переменной размещаются в стеке

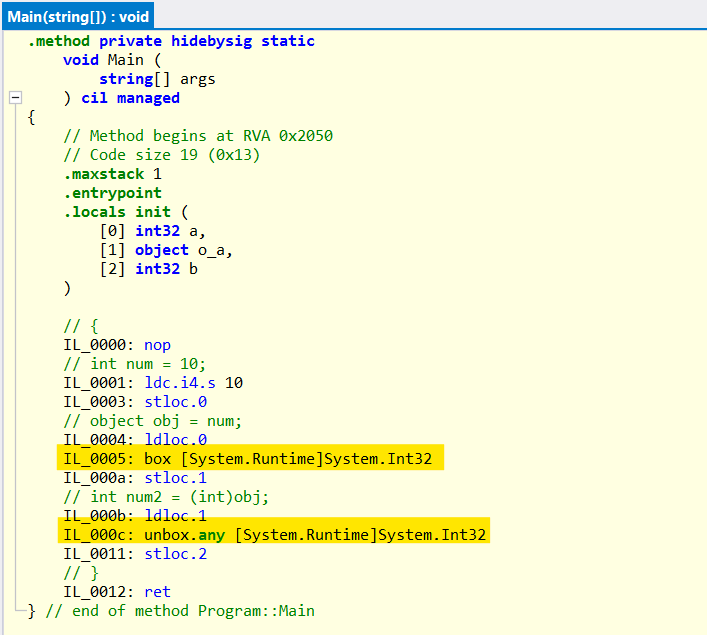
object o\_a = a; //упаковка (object - базовый класс для всех типов данных в c#, т.к. object - класс, то его данные размечаются в куче)

//таким образом в переменной o\_a получаем ссылку на данные в куче

int b = (int)o\_a; //распаковка

****



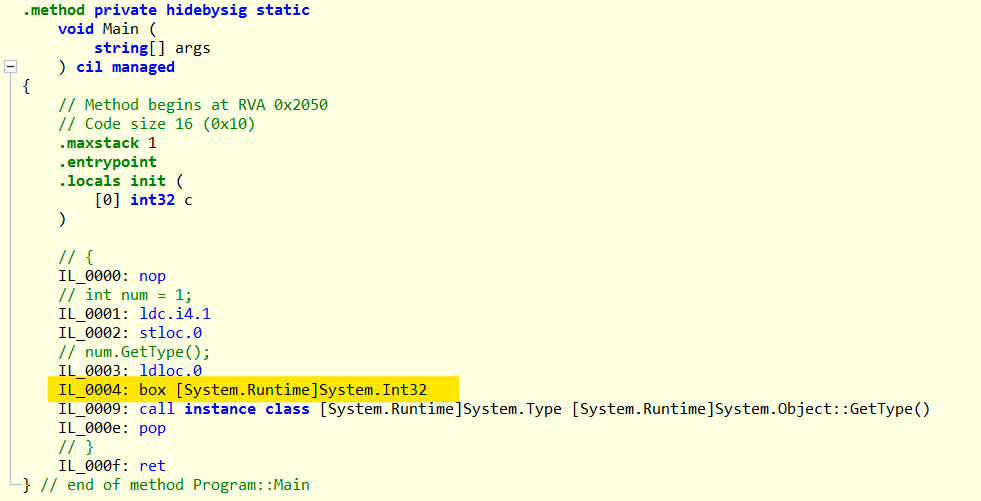
****

**//неявная упаковка/распаковка**

int c = 1;

c.GetType(); // возвращает тип данных переменной c

Вызовом метода мы значимый тип данных разместили в куче.

****

Несмотря на все недостатки в плане падения производительности .NET -приложения, концепции упаковки и распаковки были внедрены в .NET не просто так. И вот причины:

* .NET-стандарт обладает общей системой типов, что позволяет представлять и ссылочные, и значимые типы схожим образом - и все это благодаря упаковке.
* Коллекции можно было использовать для хранения значимых типов до появления обобщений.
* Упрощения кода, вроде конкатенации строк и так далее.

Упаковка и распаковка настолько распространены, что мы не можем избежать их полностью. Мы должны знать принцип их работы, чтобы минимизировать их использование, но к этому нужно подходить разумно. Не тратьте свое время на постоянную оптимизацию кода, частую проверку через IL, чтобы убедиться, дабы ни одна лишняя операция упаковки не была использована. Помните, что чистота и простота чтения кода иногда значительно более важна, нежели незаметное, мельчайшее ускорение работы программы.

**2(d) Продемонстрируйте работу с неявно типизированной переменной.**

Ранее мы явным образом указывали тип переменных, например, int x;. И компилятор при запуске уже знал, что x хранит целочисленное значение.

Однако мы можем использовать и модель неявной типизации:

|  |  |
| --- | --- |
| 1  2  3  4  5 | var hello = "Hell to World";  var c = 20;    Console.WriteLine(c.GetType().ToString());  Console.WriteLine(hello.GetType().ToString()); |

Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. В примере выше использовалось выражение Console.WriteLine(c.GetType().ToString());, которое позволяет нам узнать выведенный тип переменной с. Так как по умолчанию все целочисленные значения рассматриваются как значения типа int, то в итоге переменная c будет иметь тип int или System.Int32

Эти переменные подобны обычным, однако они имеют некоторые **ограничения**.

Во-первых, мы не можем сначала объявить неявно типизируемую переменную, а затем инициализировать:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // этот код работает  int a;  a = 20;    // этот код не работает  var c;  c = 20; |

Во-вторых, мы не можем указать в качестве значения неявно типизируемой переменной null:

|  |  |
| --- | --- |
| 1  2 | // этот код не работает  var c = null; |

Так как значение null, то компилятор не сможет вывести тип данных.

Также Вы не можете использовать var в качестве типа параметра или возвращаемого значения метода.

**КОД:**

//var - ключевое слово, которое позволяет компилятору выяснить тип переменной

var a = 5;

var b = "I was born on the wrong side of the train tracks";

var c = 9.87f;

**2(e) Продемонстрируйте пример работы с Nullable переменной**

Значение null по умолчанию могут принимать только объекты ссылочных типов. Однако в различных ситуациях бывает удобно, чтобы объекты числовых типов данных имели значение null, то есть были бы не определены. Стандартный пример - работа с базой данных, которая может содержать значения null. И мы можем заранее не знать, что мы получим из базы данных – какое-то определенное значение или же null. Для этого надо использовать знак вопроса **?** после типа значений. Например:

|  |  |
| --- | --- |
| 1  2 | int? z = null;  bool? enabled = null; |

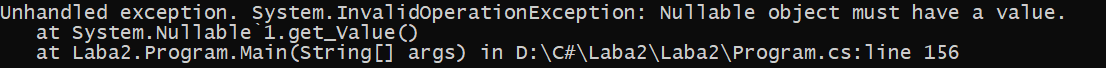
Но фактически запись **?** является упрощенной формой использования структуры **System.Nullable<T>**. Параметр **T** в угловых скобках представляет универсальный параметр, вместо которого в конкретной задаче уже подставляется конкретный тип данных. Следующие виды определения переменных будут эквивалентны:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int? z1 = 5;  bool? enabled1 = null;  Double?  d1 = 3.3;    Nullable<int> z2 = 5;  Nullable<bool> enabled2 = null;  Nullable<System.Double> d2 = 3.3; |

Для всех типов Nullable определено два свойства: **Value**, которое представляет значение объекта, и **HasValue**, которое возвращает true, если объект Nullable хранит некоторое значение.

Если мы попробуем использовать свойство Value для переменной со значением null, то получим ошибку.

Console.WriteLine($"x.Value : {x.Value}"); //приведет к ошибке, так как x = null



Поэтому нужно выполнять проверку:

int? x = null;

if(x.HasValue)

    Console.WriteLine(x.Value);

else

    Console.WriteLine("x is equal to null");

При проверке объектов на равенство следует учитывать, что они равны не только, когда они имеют ненулевые значения, которые совпадают, но и когда оба объекта равны null:

int? a = null;

int? b = null;

Console.WriteLine($"a = null, b = null \n" +

$"a == b : {a == b}"); //True

**КОД:**

//свойства Nullable-типа

int? x = null;

Console.WriteLine($"x.HasValue : {x.HasValue}"); //x = null, значит выведет False

Console.WriteLine($"x ?? \"-1\" : {x ?? -1}"); // если x - null, выведет -1, иначе значение x

//Console.WriteLine($"x.Value : {x.Value}"); //приведет к ошибке, так как x = null

Console.WriteLine($"x : {x}"); //выведет пустоту

//математические операции с Nullable-типом

x = 2;

int? y = null;

Console.WriteLine($"\nx = 2, y = null \n" +

$"x + 3 = {x + 3} \n" + //5

$"x < 1 : {x < 1} \n" + //False

$"x + y = {x + y}"); //number "+,-,/,\*" null = null

//объекты равны, когда их значения null

int? a = null;

int? b = null;

Console.WriteLine($"\na = null, b = null \n" +

$"a == b : {a == b}"); //True

//явное приведение Nullable-типа

float f\_a = (float)x; //если Nullable-тип имеет значение отличное от null, то мы можем преобразовать его к другому типу по правилам преобразования

Console.WriteLine("\n" + f\_a);

**1(f) Определите переменную типа var и присвойте ей любое значение. Затем следующей инструкцией присвойте ей значение другого типа. Объясните причину ошибки.**

**КОД:**

var w = 10;

w = "Every single one\'s got a story to tell";

Мы не можем так сделать, ибо ключевое слово var может представлять любой тип, и какой это будет **тип определяется во время компиляции**. Соответственно мы не можем просто взять и менять тип по ходу работы программы.

**СТРОКИ**

**2(a) Объявите строковые литералы. Сравните их.**

Для сравнения строк применяется статический метод Compare:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | string s1 = "hello";  string s2 = "world";    int result = String.Compare(s1, s2);  if (result<0)  {      Console.WriteLine("Строка s1 перед строкой s2");  }  else if (result > 0)  {      Console.WriteLine("Строка s1 стоит после строки s2");  }  else  {      Console.WriteLine("Строки s1 и s2 идентичны");  }  // результатом будет "Строка s1 перед строкой s2" |

Данная версия метода Compare принимает две строки и возвращает число. Если первая строка по алфавиту стоит выше второй, то возвращается число меньше нуля. В противном случае возвращается число больше нуля. И третий случай - если строки равны, то возвращается число 0.

Ещё можно сравнить строки обычным “==”:

string a = "abc";

string b = "abc";

Console.WriteLine(a == b);

b = "abb";

Console.WriteLine(a == b);



**КОД:**

string st1 = "Ito";

string st2 = "Mitsu";

//Для сравнения строк применяется статический метод Compare

//Если первая строка по алфавиту стоит выше второй, то возвращается число меньше нуля. В противном случае возвращается число больше нуля.

//И третий случай - если строки равны, то возвращается число 0.

Console.WriteLine(String.Compare(st1, st2)); //-1

**2(b) Создайте три строки на основе String. Выполните: сцепление, копирование, выделение подстроки, разделение строки на слова, вставки подстроки в заданную позицию, удаление заданной подстроки. Продемонстрируйте интерполирование строк.**

**Конкатенация**

Конкатенация строк или объединение может производиться как с помощью операции +, так и с помощью метода Concat:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | string s1 = "hello";  string s2 = "world";  string s3 = s1 + " " + s2; // результат: строка "hello world"  string s4 = String.Concat(s3, "!!!"); // результат: строка "hello world!!!"    Console.WriteLine(s4); |

Метод Concat является статическим методом класса String, принимающим в качестве параметров две строки. Также имеются другие версии метода, принимающие другое количество параметров.

Для объединения строк также может использоваться метод Join:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string s5 = "apple";  string s6 = "a day";  string s7 = "keeps";  string s8 = "a doctor";  string s9 = "away";  string[] values = new string[] { s5, s6, s7, s8, s9 };    String s10 = String.Join(" ", values);  // результат: строка "apple a day keeps a doctor away" |

Метод Join также является статическим. Использованная выше версия метода получает два параметра: строку-разделитель (в данном случае пробел) и массив строк, которые будут соединяться и разделяться разделителем.

**Копирование** **строки**

String.Copy(String) создает новый экземпляр String с тем же значением, что и указанная String .

string s1 = "Broken boy, ";

string copy\_method = String.Copy(s1);

Можно также использовать метод Substring (см. далее):

string sub\_method = s1.Substring(0, s1.Length);

**Выделение подстроки**

Обрезать определенную часть строки позволяет функция **Substring**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string text = "Хороший день";  // обрезаем, начиная с третьего символа  text = text.Substring(2);  // результат "роший день"  Console.WriteLine(text);  // обрезаем сначала до последних двух символов  text = text.Substring(0, text.Length - 2);  // результат "роший де"   Console.WriteLine(text); |

Функция Substring также возвращает обрезанную строку. В качестве параметра первая использованная версия применяет индекс, начиная с которого надо обрезать строку. Вторая версия применяет два параметра - индекс начала обрезки и **длину** вырезаемой части строки.

**Разделение строк**

С помощью функции Split мы можем разделить строку на массив подстрок. В качестве параметра функция Split принимает массив символов или строк, которые и будут служить разделителями.

string s1 = "Broken boy, ";

string s2 = "How does ";

string s3 = "it feel?";

string concat\_plus = s1 + s2 + s3;

string[] words\_arr = concat\_plus.Split(" ");

foreach(string w in words\_arr)

{

Console.WriteLine(w);

}

**Вставка**

Для вставки одной строки в другую применяется функция Insert:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string text = "Хороший день";  string subString = "замечательный ";    text = text.Insert(8, subString);  Console.WriteLine(text); |

Первым параметром в функции Insert является индекс, по которому надо вставлять подстроку, а второй параметр - собственно подстрока.

**Удаление строк**

Удалить часть строки помогает метод Remove:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | string text = "Хороший день";  // индекс последнего символа  int ind = text.Length - 1;  // вырезаем последний символ  text = text.Remove(ind);  Console.WriteLine(text);    // вырезаем первые два символа  text = text.Remove(0, 2); |

Первая версия метода Remove принимает индекс в строке, начиная с которого надо удалить все символы. Вторая версия принимает еще один параметр - **сколько символов надо удалить**.

**Интерполяция строк**

Специальный знак $ идентифицирует строковый литерал как интерполированную строку. **Интерполированная строка** — это строковый литерал, который может содержать выражения интерполяции. При разрешении интерполированной строки в результирующую элементы с выражениями интерполяции заменяются строковыми представлениями результатов выражений. Эта функция доступна начиная с C# 6.

Интерполяция строк предоставляет более понятный и удобный синтаксис для создания форматированных строк по сравнению с функцией [составного форматирования строк](https://docs.microsoft.com/ru-ru/dotnet/standard/base-types/composite-formatting). В следующем примере обе этих функции используются для получения одинаковых выходных данных:

string name = "Mark";

var date = DateTime.Now;

// Composite formatting:

Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name, date.DayOfWeek, date);

// ***String interpolation***:

Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm} now.");

// Both calls produce the same output that is similar to:

// Hello, Mark! Today is Wednesday, it's 19:40 now.

**КОД:**

string s1 = "Broken boy, ";

string s2 = "How does ";

string s3 = "it feel?";

string concat\_plus = s1 + s2 + s3;

string concat\_method = String.Concat(s1, s2, s3);

string concat\_join = String.Join("", s1, s2, s3);

Console.WriteLine("\*\*\*Concatenation\*\*\*\n" +

$"\"+\" operation : {concat\_plus}\n" +

$"method Concat : {concat\_method}\n" +

$"method Join : {concat\_join}\n");

string copy\_method = String.Copy(s1);

string assignment = s1; //приравнивание, не уверена, что так можно копировать !!!!

string sub\_method = s1.Substring(0, s1.Length);

Console.WriteLine($"\*\*\*Coping\*\*\*\n" +

$"method Copy : {copy\_method}\n" +

$"assignment : {assignment}\n" +

$"method Substring : {sub\_method}\n");

string substr1 = s3.Substring(s3.Length - 1, 1);

string substr2 = s1.Substring(7, 3);

Console.WriteLine($"\*\*\*Substring\*\*\*\n" +

$"method Substring with 1 parameter : {substr1}\n" +

$"method Substring with 2 parameters : {substr2}\n");

string[] words\_arr = concat\_plus.Split(" ");

Console.WriteLine($"\*\*\*Split\*\*\*");

foreach (string w in words\_arr)

{

Console.WriteLine(w);

}

string insert\_method = concat\_plus.Insert(1, "...");

Console.WriteLine($"\n\*\*\*Substring in position\*\*\*\n" +

$"method Insert : {insert\_method}\n");

string rm\_method1 = s1.Remove(s1.Length - 2);

string rm\_method2 = rm\_method1.Remove(0, 7);

Console.WriteLine($"\*\*\*Remove substring\*\*\*\n" +

$"method Remove with 1 parameter : {rm\_method1}\n" +

$"method Remove with 2 parameters : {rm\_method2}\n");

/\*Специальный знак $ идентифицирует строковый литерал как интерполированную строку.

\*Интерполированная строка — это строковый литерал, который может содержать выражения интерполяции.

\*При разрешении интерполированной строки в результирующую элементы с выражениями интерполяции заменяются

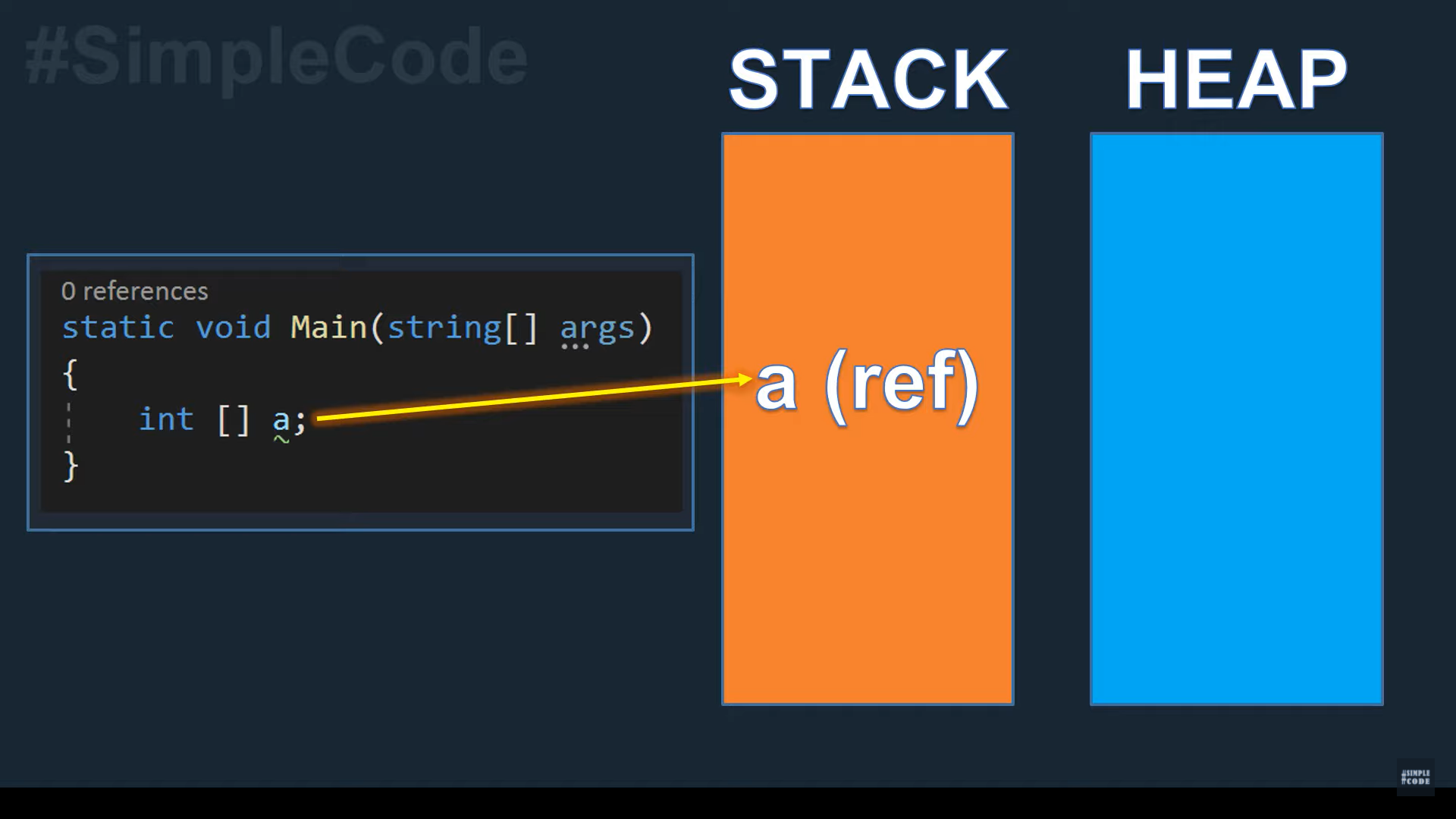
\*строковыми представлениями результатов выражений. Эта функция доступна начиная с C# 6.\*/

string interp = $"s1 : {s1}, s2 : {s2}, s3 : {s3}";

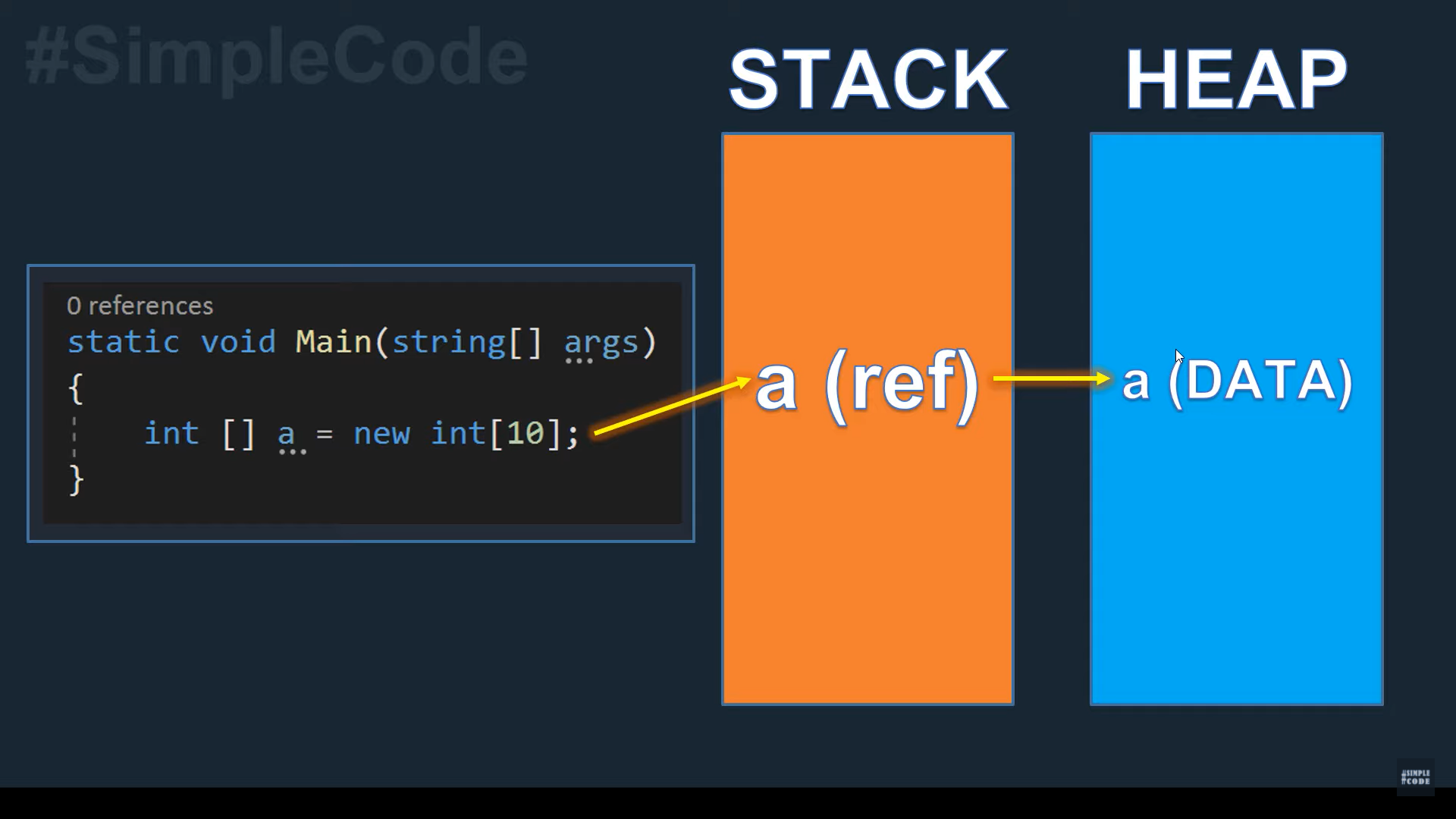
Console.WriteLine($"\*\*\*String interpolation\*\*\*\n {interp}");

**2(c) Создайте пустую и null строку. Продемонстрируйте использование метода string.IsNullOrEmpty. Продемонстрируйте, что еще можно выполнить с такими строками**

Ссылочные типы могут принимать значение **null**. Это значит, что на инициализированную переменную в стеке помещен ярлык, но он не хранит ссылку на данные в куче, так как данных нет.



При этом этому ярлыку всегда можно присвоить значение в виде ссылки на данные в куче:



# String.IsNullOrEmpty(value) Метод

Указывает, действительно ли строка value является строкой null или пустой строкой ("").

Возвращает true, если параметр value равен null или пустой строке (""); в противном случае — значение false.

string s1 = "abcd";

string s2 = "";

string s3 = null;

Console.WriteLine("String s1 {0}.", Test(s1));

Console.WriteLine("String s2 {0}.", Test(s2));

Console.WriteLine("String s3 {0}.", Test(s3));

String Test(string s)

{

if (String.IsNullOrEmpty(s))

return "is null or empty";

else

return String.Format("(\"{0}\") is neither null nor empty", s);

}

// The example displays the following output:

// String s1 ("abcd") is neither null nor empty.

// String s2 is null or empty.

// String s3 is null or empty.

**КОД:**

string n\_str = null;

string emp\_str = "";

string str = "string with text";

Console.WriteLine($"IsNullOrEmpty method\n" +

$"string is null : {String.IsNullOrEmpty(n\_str)}\n" +

$"string is empty : {String.IsNullOrEmpty(emp\_str)}\n" +

$"string with text : {String.IsNullOrEmpty(str)}\n");

Console.WriteLine($"Plus : {n\_str + emp\_str}"); //""

Console.WriteLine($"Compare : {String.Compare(n\_str, emp\_str)}"); //-1

Console.WriteLine($"String.Equals : {String.Equals(n\_str, emp\_str)}"); //False

Console.WriteLine($"String.Concat : {String.Concat(n\_str, emp\_str)}"); //""

Console.WriteLine($"Interpolation : {n\_str}, {emp\_str}"); //, ""

**2(d) Создайте строку на основе StringBuilder. Удалите определенные позиции и добавьте новые символы в начало и конец строки.**

## Класс StringBuilder

Пространство имен: [System.Text](https://docs.microsoft.com/ru-ru/dotnet/api/system.text?view=net-5.0)

Хотя класс System.String предоставляет нам широкую функциональность по работе со строками, все-таки он имеет свои недостатки. Прежде всего, объект String представляет собой *неизменяемую строку*. Когда мы выполняем какой-нибудь метод класса String, система создает **новый** объект в памяти с выделением ему достаточного места. Удаление исходной строки – не самая затратная операция. Однако когда подобных операций множество, а объем текста, для которого надо выполнить данные операции, также не самый маленький, то издержки при потере производительности становятся более существенными.

Чтобы выйти из этой ситуации во фреймворк .NET был добавлен новый класс **StringBuilder**, который находится в пространстве имен System.Text. Этот класс представляет *динамическую строку*.

При создании строки StringBuilder выделяет памяти больше, чем необходимо этой строке:

|  |  |
| --- | --- |
| 1  2  3 | StringBuilder sb = new StringBuilder("Привет мир");  Console.WriteLine($"Длина строки: {sb.Length}");  Console.WriteLine($"Емкость строки: {sb.Capacity}", ); |

Теперь переменная sb представляет начальную строку "Привет мир". Эта строка имеет длину в 10 символов. Для хранения длины в классе StringBuilder есть свойство Length. Однако есть и вторая величина - емкость выделенной памяти. Это значение хранится в свойстве Capacity. **Емкость** – это выделенная под объект память. Хотя в данном случае длина равна 10 символов, но реально емкость будет составлять по умолчанию 16 символов.

Класс StringBuilder имеет еще ряд конструкторов, которые позволяют разными способами выполнить начальную инициализацию объекта. Так, мы можем задать пустой объект, но установить начальную емкость:

|  |  |
| --- | --- |
| 1 | StringBuilder sb = new StringBuilder(20); |

Если у нас заранее известен максимальный размер объекта, мы можем таким образом сразу задать емкость и избежать последующих издержек при дополнительном выделении памяти.

Теперь посмотрим на примере использование и преимущества класса StringBuilder:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | StringBuilder sb = new StringBuilder("Название: ");  Console.WriteLine($"Длина строки: {sb.Length}"); // 10  Console.WriteLine($"Емкость строки: {sb.Capacity}"); // 16    sb.Append(" Руководство");  Console.WriteLine($"Длина строки: {sb.Length}"); // 22  Console.WriteLine($"Емкость строки: {sb.Capacity}"); // 32    sb.Append(" по C#");  Console.WriteLine($"Длина строки: {sb.Length}"); // 28  Console.WriteLine($"Емкость строки: {sb.Capacity}"); // 32 |

При создании объекта StringBuilder выделяется память по умолчанию для 16 символов, так как длина начальной строки меньше 16.

Дальше применяется метод **Append** - этот метод добавляет к строке подстроку. Так как при объединении строк их общая длина - 22 символа - превышает начальную емкость в 16 символов, то начальная емкость удваивается - до 32 символов.

Если бы итоговая длина строки была бы больше 32 символов, то емкость расширялась бы до размера длины строки.

Далее опять применяется метод Append, однако финальная длина уже будет 28 символов, что меньше 32 символов, и дополнительная память не будет выделяться.

Класс StringBuilder предлагает ряд методов для операций над строками:

* **Insert**: вставляет подстроку в объект StringBuilder, начиная с определенного индекса
* **Remove**: удаляет определенное количество символов, начиная с определенного индекса
* **Replace**: заменяет все вхождения определенного символа или подстроки на другой символ или подстроку
* **AppendFormat()** Добавляет строку, сформированную в соответствии со спецификатором формата
* **Append()** Добавляет строку к текущей строке
* **ToString()** Возвращает текущую строку в виде объекта System.String (переопределение метода класса System.Object)

Используем методы StringBuilder:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | StringBuilder sb = new StringBuilder("Привет мир");  sb.Append("!");  sb.Insert(7, "компьютерный ");  Console.WriteLine(sb);    // заменяем слово  sb.Replace("мир", "world");  Console.WriteLine(sb);    // удаляем 13 символов, начиная с 7-го  sb.Remove(7, 13);  Console.WriteLine(sb);    // получаем строку из объекта StringBuilder  string s = sb.ToString();  Console.WriteLine(s); |

Когда надо использовать класс String, а когда StringBulder?

Microsoft рекомендует использовать класс String в следующих случаях:

* При небольшом количестве операций и изменений над строками
* При выполнении фиксированного количества операций объединения. В этом случае компилятор может объединить все операции объединения в одну
* Когда надо выполнять масштабные операции поиска при построении строки, например IndexOf или StartsWith. Класс StringBuilder не имеет подобных методов.

Класс StringBuilder рекомендуется использовать в следующих случаях:

* При неизвестном количестве операций и изменений над строками во время выполнения программы
* Когда предполагается, что приложению придется сделать множество подобных операций

**КОД:**

StringBuilder str = new StringBuilder("Can you please sit the fuck down"); //выделяет больше памяти, чем нужно в данный момент, для возможности менять строку

Console.WriteLine($"Remove : {str.Remove(0, 23)}");

str.Insert(0, "->");

str.Append("<-");

Console.WriteLine($"Insert + Append: {str}");

**МАССИВЫ**

**3(a) Создайте целый двумерный массив и выведите его на консоль в отформатированном виде (матрица).**

## Массивы

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

|  |  |
| --- | --- |
| 1 | тип\_переменной[] название\_массива; |

Например, определим массив целых чисел:

|  |  |
| --- | --- |
| 1 | int[] numbers; |

После определения переменной массива мы можем присвоить ей значение:

|  |  |
| --- | --- |
| 1 | int[] nums = new int[4]; |

Здесь вначале мы объявили массив nums, который будет хранить данные типа int. Далее используя операцию new, мы выделили память для 4 элементов массива: new int[4]. Число 4 еще называется **длиной массива**. При таком определении все элементы получают значение по умолчанию, которое предусмотренно для их типа. Для типа int значение по умолчанию - 0.

Также мы сразу можем указать значения для этих элементов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | int[] nums2 = new int[4] { 1, 2, 3, 5 };    int[] nums3 = new int[] { 1, 2, 3, 5 };    int[] nums4 = new[] { 1, 2, 3, 5 };    int[] nums5 = { 1, 2, 3, 5 }; |

Все перечисленные выше способы будут равноценны.

Для обращения к элементам массива используются **индексы**. Индекс представляет номер элемента в массиве, при этом нумерация начинается с нуля, поэтому индекс первого элемента будет равен 0. А чтобы обратиться к четвертому элементу в массиве, нам надо использовать индекс 3, к примеру: nums[3]. Используем индексы для получения и установки значений элементов массива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int[] nums = new int[4];  nums[0] = 1;  nums[1] = 2;  nums[2] = 3;  nums[3] = 5;  Console.WriteLine(nums[3]);  // 5 |

И так как у нас массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу: nums[5] = 5;. Если мы так попытаемся сделать, то мы получим исключение **IndexOutOfRangeException**.

### Перебор массивов. Циклы foreach и for

**Цикл foreach** предназначен для перебора элементов в контейнерах, в том числе в массивах. Формальное объявление цикла foreach:

|  |  |
| --- | --- |
| 1  2  3  4 | foreach (тип\_данных название\_переменной in контейнер)  {      // действия  } |

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = new int[] { 1, 2, 3, 4, 5 };  foreach (int i in numbers)  {      Console.WriteLine(i);  } |

Здесь в качестве контейнера выступает массив данных типа int. Поэтому мы объявляем переменную с типом int

Подобные действия мы можем сделать и с помощью **цикла for**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | int[] numbers = new int[] { 1, 2, 3, 4, 5 };  for (int i = 0; i < numbers.Length; i++)  {      Console.WriteLine(numbers[i]);  } |

В то же время цикл for более гибкий по сравнению с foreach. Если foreach последовательно извлекает элементы контейнера и только для чтения, то в цикле for мы можем перескакивать на несколько элементов вперед в зависимости от приращения счетчика, а также можем изменять элементы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int[] numbers = new int[] { 1, 2, 3, 4, 5 };  for (int i = 0; i < numbers.Length; i++)  {      numbers[i] = numbers[i] \* 2;      Console.WriteLine(numbers[i]);  } |

### Многомерные массивы

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 горизонтального ряда 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. Например, создадим одномерный и двухмерный массивы, которые имеют одинаковые элементы:

|  |  |
| --- | --- |
| 1  2  3 | int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };    int[,] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } }; |

Визуально оба массива можно представить следующим образом:

##### Одномерный массив nums1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 |

##### Двухмерный массив nums2

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| 3 | 4 | 5 |

Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Все возможные способы определения двухмерных массивов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | int[,] nums1;  int[,] nums2 = new int[2, 3];  int[,] nums3 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };  int[,] nums4 = new int[,] { { 0, 1, 2 }, { 3, 4, 5 } };  int[,] nums5 = new [,]{ { 0, 1, 2 }, { 3, 4, 5 } };  int[,] nums6 = { { 0, 1, 2 }, { 3, 4, 5 } }; |

Первое число в квадратных скобках – количество подмассивов, второе – длина этих подмассивов.

Массивы могут иметь и большее количество измерений. Объявление трехмерного массива могло бы выглядеть так:

|  |  |
| --- | --- |
| 1 | int[,,] nums3 = new int[2, 3, 4]; |

Соответственно могут быть и четырехмерные массивы и массивы с большим количеством измерений. Но на практике обычно используются одномерные и двухмерные массивы.

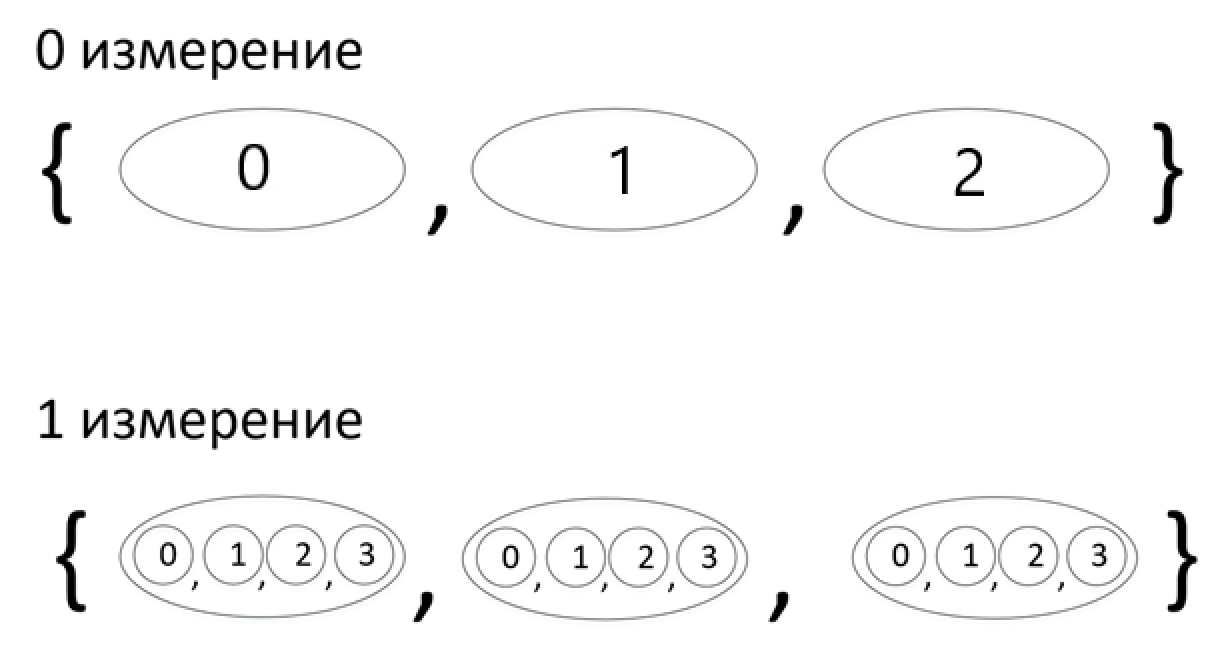
Определенную сложность может представлять перебор многомерного массива. Прежде всего, надо учитывать, что длина такого массива - это совокупное количество элементов.

|  |  |
| --- | --- |
| 1  2  3  4 | int[,] mas = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };  foreach (int i in mas)      Console.Write($"{i} ");  Console.WriteLine(); |

В данном случае длина массива mas равна 12. И цикл foreach выводит все элементы массива в строку:

1 2 3 4 5 6 7 8 9 10 11 12

Но что если мы хотим отдельно пробежаться по каждой строке в таблице? В этом случае надо получить количество элементов в размерности. В частности, у каждого массива есть метод GetUpperBound(dimension), который возвращает индекс последнего элемента в определенной размерности. И если мы говорим непосредственно о двухмерном массиве, то первая размерность (с индексом 0) строки. И с помощью выражения mas.GetUpperBound(0) + 1 можно получить количество строк таблицы, представленной двухмерным массивом.



А через  mas.Length / rows можно получить количество элементов в каждой строке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | int[,] mas = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } };    int rows = mas.GetUpperBound(0) + 1;  int columns = mas.Length / rows;  // или так  // int columns = mas.GetUpperBound(1) + 1;    for (int i = 0; i < rows; i++)  {      for (int j = 0; j < columns; j++)      {          Console.Write($"{mas[i, j]} \t");      }      Console.WriteLine();  } |

1 2 3

4 5 6

7 8 9

10 11 12

**Основные понятия массивов**

Суммирую основные понятия массивов:

* **Ранг** (rank): количество измерений массива
* **Длина измерения** (dimension length): длина отдельного измерения массива
* **Длина массива** (array length): количество всех элементов массива

Например, возьмем массив

|  |  |
| --- | --- |
| 1 | int[,] numbers = new int[3, 4]; |

Массив numbers двухмерный, то есть он имеет два измерения, поэому его ранг равен 2. Длина первого измерения - 3, длина второго измерения - 4. Длина массива (то есть общее количество элементов) - 12.

**КОД:**

int[,] nums = { { 0, 1, 2, 3 }, { 4, 5, 6, 7 }, { 8, 9, 10, 11 } };

int rows = nums.GetUpperBound(0) + 1;

int cols = nums.GetUpperBound(1) + 1;

for(int i = 0; i < rows; i++)

{

for(int j = 0; j < cols; j++)

{

Console.Write(nums[i, j] + "\t");

}

Console.WriteLine("\n");

}

**3(b) Создайте одномерный массив строк. Выведите на консоль его содержимое, длину массива. Поменяйте произвольный элемент (пользователь определяет позицию и значение).**

**КОД:**

string[] strs = new string[6] { "I was born on the wrong side of the train tracks",

"I was raised with a strap across my back",

"Lay me on my side or hold me up to the light, yeah",

"I was burned by the cold kiss of a vampire",

"I was bit by the whisper of a soft liar",

"Any good friend of yours is a good friend of mine"};

Console.WriteLine("\*\*\*Array\*\*\*");

foreach (string s in strs)

{

Console.WriteLine(s);

}

Console.WriteLine($"\nArray's length : {strs.Length}\n");

Console.Write("Введите номер строки : ");

int ind = Convert.ToInt32(Console.ReadLine()) - 1;

//если введенное число больше, количсетва строк в массиве, то менять будем последнюю строку

if (ind >= strs.Length) {

ind = strs.Length - 1;

}

Console.Write("Введите, на что заменить эту строку : ");

string new\_str = Console.ReadLine();

strs[ind] = new\_str;

Console.WriteLine("\n\*\*\*Array witn changed line\*\*\*");

foreach (string s in strs)

{

Console.WriteLine(s);

}

**3(c) Создайте ступечатый (не выровненный) массив вещественных чисел с 3-мя строками, в каждой из которых 2, 3 и 4 столбцов соответственно. Значения массива введите с консоли.**

Двумерный массив можно представить в виде таблицы, в которой длина каждой строки остается неизменной по всему массиву. Но в C# можно также создавать специальный тип двумерного массива, называемый *ступенчатым массивом*. **Ступенчатый массив** представляет собой массив массивов, в котором длина каждого массива может быть *разной*. Следовательно, ступенчатый массив может быть использован для составления таблицы из строк разной длины.

Ступенчатые массивы объявляются с помощью ряда квадратных скобок, в которых указывается их размерность. Например, для объявления двумерного ступенчатого массива служит следующая общая форма:

***тип [][] имя массива = new тип[размер] [];***

где размер обозначает число строк в массиве. Память для самих строк распределяется индивидуально, и поэтому длина строк может быть разной.

Давайте рассмотрим пример использования ступенчатого массива:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication1

{

class Program

{

static void Main(string[] args)

{

int i = 0;

// Объявляем ступенчатый массив

int[][] myArr = new int[4][];

myArr[0] = new int[4];

myArr[1] = new int[6];

myArr[2] = new int[3];

myArr[3] = new int[4];

// Инициализируем ступенчатый массив

for (; i < 4; i++)

{

myArr[0][i] = i;

Console.Write("{0}\t",myArr[0][i]);

}

Console.WriteLine();

for (i = 0; i < 6; i++)

{

myArr[1][i] = i;

Console.Write("{0}\t", myArr[1][i]);

}

Console.WriteLine();

for (i = 0; i < 3; i++)

{

myArr[2][i] = i;

Console.Write("{0}\t", myArr[2][i]);

}

Console.WriteLine();

for (i = 0; i < 4; i++)

{

myArr[3][i] = i;

Console.Write("{0}\t", myArr[3][i]);

}

Console.ReadLine();

}

}

}

Созданный массив в данном примере можно структурно представить в виде:



**КОД:**

float[][] arr = new float[3][];

arr[0] = new float[2];

arr[1] = new float[3];

arr[2] = new float[4];

//ВВОД МАССИВА

Console.Write("Введите в массив 9 вещественных чисел через пробел : ");

//через пробел делим строку на элементы массива и записываем получившиеся строки в массив

string[] usual\_arr\_str = (Console.ReadLine().Split(" "));

int usual\_ind = 0;

for (int i = 0; i < 3; i++) //3 строки

{

int max\_col = 0;

if (i == 0) {

max\_col = 2;

}

else if(i == 1)

{

max\_col = 3;

}

else if (i == 2)

{

max\_col = 4;

}

for (int j = 0; j < max\_col; j++) //2, 3, затем 4 столбца

{

//элементы массива строк приводим к вещественным числам

arr[i][j] = Convert.ToSingle(usual\_arr\_str[usual\_ind]);

usual\_ind++;

}

}

//ВЫВОД МАССИВА

for (int i = 0; i < 3; i++)

{

Console.Write("\n");

int max\_col = 0;

if (i == 0)

{

max\_col = 2;

}

else if (i == 1)

{

max\_col = 3;

}

else if (i == 2)

{

max\_col = 4;

}

for (int j = 0; j < max\_col; j++)

{

Console.Write(arr[i][j] + "\t");

}

Console.WriteLine("\n");

}

**3(d)** **Создайте неявно типизированные переменные для хранения массива и строки.**

**КОД:**

var str = "From Today on, It's My Turn!!";

var arr = new[] { 1.2, 7.09, 8.3 };

Console.WriteLine($"String : {str}");

Console.Write($"Array : ");

foreach (float i in arr)

{

Console.Write(i + "\t");

}

Console.Write("\n");

**КОРТЕЖИ**

**4(a, b)**

**Задайте кортеж из 5 элементов с типами int, string, char, string, ulong.**

**Выведите кортеж на консоль целиком и выборочно (например, 1, 3, 4 элементы)**

Кортежи предоставляют удобный способ для работы с набором значений, который был добавлен в версии C# 7.0.

Кортеж представляет набор значений, заключенных в круглые скобки:

|  |  |
| --- | --- |
| 1 | var tuple = (5, 10); |

В данном случае определен кортеж tuple, который имеет два значения: 5 и 10. В дальнейшем мы можем обращаться к каждому из этих значений через поля с названиями Item[порядковый\_номер\_поля\_в\_кортеже]. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | static void Main(string[] args)  {      var tuple = (5, 10);      Console.WriteLine(tuple.Item1); // 5      Console.WriteLine(tuple.Item2); // 10      tuple.Item1 += 26;      Console.WriteLine(tuple.Item1); // 31      Console.Read();  } |

В данном случае тип определяется неявно. Но мы ткже можем явным образом указать для переменной кортежа тип:

|  |  |
| --- | --- |
| 1 | (int, int) tuple = (5, 10); |

Так как кортеж содержит два числа, то в определении типа нам надо указать два числовых типа. Или другой пример определения кортежа:

|  |  |
| --- | --- |
| 1 | (string, int, double) person = ("Tom", 25, 81.23); |

Первый элемент кортежа в данном случае представляет строку, второй элемент - тип int, а третий - тип double.

Мы также можем дать названия полям кортежа:

|  |  |
| --- | --- |
| 1  2  3 | var tuple = (count:5, sum:10);  Console.WriteLine(tuple.count); // 5  Console.WriteLine(tuple.sum); // 10 |

Теперь чтобы обратиться к полям кортежа используются их имена, а не названия Item1 и Item2.

Мы даже можем не использовать переменную для определения всего кортежа, а использовать отдельные переменные для его полей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | static void Main(string[] args)  {      var (name, age) = ("Tom", 23);      Console.WriteLine(name);    // Tom      Console.WriteLine(age);     // 23      Console.Read();  } |

В этом случае с полями кортежа мы сможем работать как с переменными, которые определены в рамках метода.

**КОД:**

(int, string, char, string, ulong) cor = (7, "Kyou kara Ore wa!!", 'A', "Rainforest Alliance", 45);

Console.WriteLine(cor);

Console.WriteLine($"\n1 cortege element : {cor.Item1}");

Console.WriteLine($"2 cortege element : {cor.Item2}");

Console.WriteLine($"3 cortege element : {cor.Item3}");

Console.WriteLine($"4 cortege element : {cor.Item4}");

Console.WriteLine($"5 cortege element : {cor.Item5}");

**4(c)** **Выполните распаковку кортежа в переменные. Продемонстрируйте различные способы распаковки кортежа. Продемонстрируйте использование переменной ( \_ ). (доступно начиная с C#7.3)**

C # имеет встроенную поддержку деконструкции кортежей, которая позволяет распаковать все элементы в кортеже за одну операцию. Общий синтаксис для деконструкции кортежа аналогичен синтаксису для его определения: вы заключаете переменные, которым должен быть присвоен каждый элемент, в круглые скобки в левой части оператора присваивания.

**КОД:**

//явное объявление первых двух переменных и ключевое слово var, чтобы среда сама определила тип третьей переменной

(string song, int num, var fl) = ("Smells Like Teen Spirit", 10, 9.4f);

//среда объявлет тип всех переменных

var (int\_num, ch) = (-10, 'E');

//вы можете разложить кортеж на уже объявленные переменные

int a;

int b = 9;

(a, b) = (10, 7);

**4(d) Сравните два кортежа.**

► Начиная с C # 7.3, кортежи поддерживают операторы ==и !=. Эти операторы сравнивают элементы левого операнда с соответствующими элементами правого операнда, следуя порядку элементов кортежа.

(int a, byte b) left = (5, 10);

(long a, int b) right = (5, 10);

Console.WriteLine(left == right); // output: True

Console.WriteLine(left != right); // output: False

var t1 = (A: 5, B: 10);

var t2 = (B: 5, A: 10);

Console.WriteLine(t1 == t2); // output: True

Console.WriteLine(t1 != t2); // output: False

Как в предыдущем примере, показывает, что ==и !=операции не учитывают имена кортежа поля.

Два кортежа сравнимы, если выполняются оба следующих условия:

* Оба кортежа имеют одинаковое количество элементов. Например, t1 != t2 не компилируется if t1 и t2 имеет разное количество элементов.
* Для каждой позиции кортежа, соответствующие элементы из левых и правых кортежей операндов сравнимы с == и != операторами. Например, (1, (2, 3)) == ((1, 2), 3) не компилируется, потому что 1 не сравнимо с (1, 2).

Операторы ==и !=сравнивают кортежи коротким замыканием. То есть операция останавливается, как только встречает пару неравных элементов или достигает концов кортежей.

► Метод **CompareTo** сравнивает текущий кортеж с заданным объектом и возвращает целое число, указывающее, находится ли текущий объект в той же позиции, что и указанный объект, после него или перед ним в порядке сортировки.

int IComparable.CompareTo (object obj);

#### Параметры

**obj –** Объект для сравнения с текущим экземпляром.

#### Возвращаемое значение

Целое число со знаком, определяющее относительное положение экземпляра и параметра obj в порядке сортировки, как показано в следующей таблице.

| **ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ** | |
| --- | --- |
| **Значение** | **Описание** |
| Отрицательное целое число | Данный экземпляр предшествует параметру obj. |
| Нуль | У этого экземпляра та же позиция в порядке сортировки, что и у obj. |
| Положительное целое число | Данный экземпляр стоит после параметра obj. |

► Метод **Equals** возвращает значение, показывающее, равен ли текущий объект указанному объекту. Возвращаемое значение true, если текущий экземпляр равен указанному объекту; в противном случае — false. Позволяет сравнивать кортежи с разным количеством элементов.

**КОД:**

(int, int, int, int, int) cor1 = (10, 20, 9, 0, 40);

(int, int, int, int, int) cor2 = (20, 10, 9, 75, -2);

//позволяет сравнивать кортежи с разным количеством элементов

Console.WriteLine("cor2.Equals(cor1) : " + cor2.Equals(cor1)); //False

//компилируются только, если количество элементов в первом кортеже равно количеству во втором

Console.WriteLine("cor1 == cor2 : " + (cor1 == cor2)); //False

Console.WriteLine("cor2.CompareTo(cor1) : " + cor2.CompareTo(cor1)); //1

Console.WriteLine("cor1.CompareTo(cor2) : " + cor1.CompareTo(cor2)); //-1

**ЛОКАЛЬНЫЕ ФУНКЦИИ**

**Создайте локальную функцию в main и вызовите ее. Формальные параметры функции – массив целых и строка. Функция должна вернуть кортеж, содержащий: максимальный и минимальный элементы массива, сумму элементов массива и первую букву строки.**

**Локальные функции** — это новая возможность в C# 7, которая позволяет определять функцию внутри другой функции (метода).

Определим и используем локальную функцию:

class Program

{

    static void Main(string[] args)

    {

        var result = GetResult(new int[] { -3, -2, -1, 0, 1, 2, 3 });

        Console.WriteLine(result);  // 6

        Console.Read();

    }

    static int GetResult(int[] numbers)

    {

        int limit = 0;

        // локальная функция

        bool IsMoreThan(int number)

        {

            return number > limit;

        }

        int result = 0;

        for(int i=0; i < numbers.Length; i++)

        {

            if (IsMoreThan(numbers[i]))

            {

                result += numbers[i];

            }

        }

        return result;

    }

}

Здесь в методе GetResult определена локальная функция IsMoreThan(), которая может быть вызвана только внутри этого метода. Локальная функция задает еще одну область видимости, где мы можем определять переменные и выполнять над ними действия. В то же время ей доступны все переменные, которые определены в том же методе.

При использовании локальных функций следует помнить, что они не могут иметь модификаторов доступа (public, private, protected). Нельзя определить в одном методе несколько локальных функций с одним и тем же именем, даже если у них будет разный список параметров. Кроме того, не имеет значения, определены локальные функции до своего вызова или после.

Начиная с версии C# 8.0 можно определять статические локальные функции. Их особенностью является то, что они не могут обращаться к переменным окружения, то есть метода, в котором статическая функция определена. Например, перепишем метод GetResult, сделав функцию IsMoreThan статической:

static int GetResult(int[] numbers)

{

    // статическая локальная функция

    static bool IsMoreThan(int number)

    {

        int limit = 0;

        return number > limit;

    }

    int result = 0;

    for (int i = 0; i < numbers.Length; i++)

    {

        if (IsMoreThan(numbers[i]))

            result += numbers[i];

    }

    return result;

}

**КОД:**

(int, int, int, char) func (int[] arr,string str){

int min = arr[0];

int max = arr[0];

int sum = 0;

foreach (int num in arr)

{

if (num > max)

{

max = num;

}

if (num < min)

{

min = num;

}

sum += num;

}

return (max, min, sum, str[0]);

}

var result = func(new int[] { 8, 9, 0, -1, 45 }, "Hello there!");

Console.WriteLine("Max, min, sum, first letter : " + result);

**CHECKED/UNCHECKED**

**6**

**a. Определите две локальные функции.**

**b. Разместите в одной из них блок checked, в котором определите переменную типа int с максимальным возможным значением этого типа. Во второй функции определите блок unchecked с таким же содержимым.**

**c. Вызовите две функции. Проанализируйте результат.**

В C# имеется специальное средство, связанное с генерированием исключений, возникающих при переполнении в арифметических вычислениях. Как известно, результаты некоторых видов арифметических вычислений могут превышать диапазон представления чисел для типа данных, используемого в вычислении. В этом случае происходит так называемое **переполнение**.

В C# допускается указывать, будет ли в коде сгенерировано исключение при переполнении, с помощью ключевых слов *checked* и *unchecked*. Так, если требуется указать, что выражение будет проверяться на переполнение, следует использовать ключевое слово checked, а если требуется проигнорировать переполнение — ключевое слово unchecked. При использовании ключевого слова checked приложение выбрасывает исключение о переполнении. При использовании unchecked результат усекается, чтобы не выйти за пределы диапазона представления чисел для целевого типа выражения.

У ключевого слова checked имеются две общие формы. В одной форме проверяется конкретное выражение, и поэтому она называется операторной. А в другой форме проверяется блок операторов, и поэтому она называется блочной. Ниже приведены обе формы:

checked (выражение)

checked {

// проверяемые операторы

}

где выражение обозначает проверяемое выражение. Если вычисление проверяемого выражения приводит к переполнению, то генерируется исключение OverflowException.

У ключевого слова unchecked также имеются две общие формы. В первой, операторной форме переполнение игнорируется при вычислении конкретного выражения. А во второй, блочной форме оно игнорируется при выполнении блока операторов:

unchecked (выражение)

unchecked {

// операторы, для которых переполнение игнорируется

}

где выражение обозначает конкретное выражение, при вычислении которого переполнение игнорируется. Если же в непроверяемом выражении происходит переполнение, то результат его вычисления усекается.

## Конструкция try..catch..finally (к коду ниже)

Иногда при выполнении программы возникают ошибки, которые трудно предусмотреть или предвидеть, а иногда и вовсе невозможно. Например, при передачи файла по сети может неожиданно оборваться сетевое подключение. такие ситуации называются **исключениями**. Язык C# предоставляет разработчикам возможности для обработки таких ситуаций. Для этого в C# предназначена конструкция **try...catch...finally**.

try

{

}

catch

{

}

finally

{

}

При использовании блока **try...catch..finally** вначале выполняются все инструкции в блоке **try**. Если в этом блоке не возникло исключений, то после его выполнения начинает выполняться блок **finally**. И затем конструкция try..catch..finally завершает свою работу.

Если же в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается, и среда CLR начинает искать блок **catch**, который может обработать данное исключение. Если нужный блок catch найден, то он выполняется, и после его завершения выполняется блок finally.

Если нужный блок catch не найден, то при возникновении исключения программа аварийно завершает свое выполнение.

Следует отметить, что в этой конструкции обязателен блок try. При наличии блока catch мы можем опустить блок finally:

try

{

    int x = 5;

    int y = x / 0;

    Console.WriteLine($"Результат: {y}");

}

catch

{

    Console.WriteLine("Возникло исключение!");

}

**КОД:**

void ch(int a)

{

try

{

checked

{

a = a + 8;

}

Console.WriteLine(a);

}

catch(OverflowException)

{

Console.WriteLine("Переполнение");

}

}

void unch(int a)

{

unchecked

{

a = a + 8;

}

Console.WriteLine(a);

}

int max\_int = 2147483647;

ch(max\_int);

unch(max\_int);

**КОД ВСЕЙ ЛАБЫ**

using System;

using System.Text; //для класса StringBuilder

namespace Laba2

{

class Program2

{

static void Main(string[] args)

{

//``````````````````` ЗАДАНИЕ 1 (a) ```````````````````

//хранит значение true или false (логические литералы)

bool f = false;

bool t = true;

//хранит целое число от 0 до 255 и занимает 1 байт

byte byte\_num = 255;

//хранит целое число от -128 до 127 и занимает 1 байт

sbyte sbyte\_neg\_num = 10;

sbyte sbyte\_pos\_num = -65;

// хранит целое число от -32768 до 32767 и занимает 2 байта

short sh\_neg\_num = -4000;

short sh\_pos\_num = 2768;

//хранит целое число от 0 до 65535 и занимает 2 байта

ushort ush\_num = 64590;

//хранит целое число от -2147483648 до 2147483647 и занимает 4 байта

//Все целочисленные литералы по умолчанию представляют значения типа int

int i\_hex\_num = 0xFF; //i\_hex\_num = 255

int i\_dec\_num = -999;

//хранит целое число от 0 до 4294967295 и занимает 4 байта

uint ui\_num = 80;

//хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт

long l\_neg\_num = -602;

long l\_pos\_num = 5339;

//хранит целое число от 0 до 18 446 744 073 709 551 615 и занимает 8 байт

ulong ul\_num = 0x3A; //ul\_num = 58

//хранит число с плавающей точкой от ± 1,5\*10^-45 до ± 3,4\*10^38, точность ~6-9 цифр и занимает 4 байта

//В visual studio дробное значение по умолчанию воспринимается как тип double, чтобы явно указать,

//что у нас число типа float можно использовать в конце суффикс F/f

float f\_neg\_num = -214.3f;

float f\_pos\_num = 67.19f;

//хранит число с плавающей точкой от ±5.0\*10^-324 до ±1.7\*10^308, точность ~15-17 цифр и занимает 8 байт

double d\_neg\_num = -256.16e34;

double d\_pos\_num = 45;

//хранит число с плавающей точкой от ±1.0\*10^-28 до ±7.9228\*10^28, точность 28-29 цифр и занимает 16 байт

decimal dec\_neg\_num = -34.92223m;

decimal dec\_pos\_num = 7.21m;

//хранит одиночный символ в кодировке Unicode и занимает 2 байта

char symb = 'S';

Console.WriteLine($"Bool : {f}, {t} \n" +

$"Byte : {byte\_num} \n" +

$"Sbyte : {sbyte\_neg\_num}, {sbyte\_pos\_num} \n" +

$"Short : {sh\_neg\_num}, {sh\_pos\_num} \n" +

$"Ushort : {ush\_num} \n" +

$"Int : {i\_hex\_num}, {i\_dec\_num} \n" +

$"Uint : {ui\_num} \n" +

$"Long : {l\_neg\_num}, {l\_pos\_num} \n" +

$"Ulong : {ul\_num} \n" +

$"Float : {f\_neg\_num}, {f\_pos\_num} \n" +

$"Double : {d\_neg\_num}, {d\_pos\_num} \n" +

$"Decimal : {dec\_neg\_num}, {dec\_pos\_num} \n" +

$"Char : {symb} \n");

//``````````````````` ЗАДАНИЕ 1 (b) ```````````````````

//~~~~~ неявное преобразование

byte byte\_a = 30;

int int\_a = byte\_a;

char symbol = 'A';

float int\_symb = symb;

byte byte\_b = 10;

float float\_b = byte\_b;

long long\_c = 9872;

double double\_c = long\_c;

short short\_d = -45;

int int\_d = short\_d;

Console.WriteLine($"Неявное преобразование \n" +

$"byte -> int : {byte\_a} \n" +

$"char -> float : {int\_symb} \n" +

$"byte -> float : {float\_b} \n" +

$"long -> double : {double\_c} \n" +

$"short -> int : {int\_d}");

//~~~~~ явное преобразование

sbyte sbyte\_e = -1;

byte byte\_e = (byte)sbyte\_e;

float float\_f = 9.23f;

byte byte\_f = (byte)float\_f;

double double\_g = 12.9887615213456;

float float\_g = (float)double\_g;

int int\_j = -289;

sbyte sbyte\_j = (sbyte)int\_j;

int int\_k = 10;

short short\_k = (short)int\_k;

Console.WriteLine($"\nЯвное преобразование \n" +

$"sbyte -> byte : {byte\_e} \n" +

$"float -> byte : {byte\_f} \n" +

$"double -> float : {float\_g} \n" +

$"int -> byte : {sbyte\_j} \n" +

$"int -> short : {short\_k}");

//~~~~~ класс Convert

Console.Write("Enter your name : ");

string name = Console.ReadLine();

Console.Write("Age : ");

int age = Convert.ToByte(Console.ReadLine());

Console.WriteLine($"Name : {name}, age : {age}");

//``````````````````` ЗАДАНИЕ 1 (с) ```````````````````

//явная упаковка/распаковка

int a = 10; //данные переменной размещаются в стеке

object o\_a = a; //упаковка (object - базовый класс для всех типов данных в c#, т.к. object - класс, то его данные размечаются в куче)

//таким образом в переменной o\_a получаем ссылку на данные в куче

int b = (int)o\_a; //распаковка

//неявная упаковка/распаковка

int c = 1;

c.GetType(); // возвращает тип данных переменной c

Console.Write(c + 8);

//``````````````````` ЗАДАНИЕ 1(d) ```````````````````

//var - ключевое слово, которое позволяет компилятору выяснить тип переменной

var a1d = 5;

var b1d = "I was born on the wrong side of the train tracks";

var c1d = 9.87f;

//``````````````````` ЗАДАНИЕ 1(e) ```````````````````

//свойства Nullable-типа

int? x = null;

Console.WriteLine($"x.HasValue : {x.HasValue}"); //x = null, значит выведет False

Console.WriteLine($"x ?? \"-1\" : {x ?? -1}"); // если x - null, выведет -1, иначе значение x

//Console.WriteLine($"x.Value : {x.Value}"); //приведет к ошибке, так как x = null

Console.WriteLine($"x : {x}"); //выведет пустоту

//математические операции с Nullable-типом

x = 2;

int? y = null;

Console.WriteLine($"\nx = 2, y = null \n" +

$"x + 3 = {x + 3} \n" + //5

$"x < 1 : {x < 1} \n" + //False

$"x + y = {x + y}"); //number "+,-,/,\*" null = null

//объекты равны, когда их значения null

int? a\_nullable = null;

int? b\_nullable = null;

Console.WriteLine($"\na = null, b = null \n" +

$"a == b : {a == b}"); //True

//явное приведение Nullable-типа

float f\_a = (float)x; //если Nullable-тип имеет значение отличное от null, то мы можем преобразовать его к другому типу по правиласм преобразования

Console.WriteLine("\n" + f\_a);

//``````````````````` ЗАДАНИЕ 1 (e) ```````````````````

var w = 10;

//w = "Every single one\'s got a story to tell";

//------------------------------------------------------------------------------------------------------------------------------------

//``````````````````` ЗАДАНИЕ 2 (a) ```````````````````

string st1 = "Ito";

string st2 = "Mitsu";

//Для сравнения строк применяется статический метод Compare

//Если первая строка по алфавиту стоит выше второй, то возвращается число меньше нуля. В противном случае возвращается число больше нуля.

//И третий случай - если строки равны, то возвращается число 0.

Console.WriteLine(String.Compare(st1, st2));

//``````````````````` ЗАДАНИЕ 2(b) ```````````````````

string s1 = "Broken boy, ";

string s2 = "How does ";

string s3 = "it feel?";

string concat\_plus = s1 + s2 + s3;

string concat\_method = String.Concat(s1, s2, s3);

string concat\_join = String.Join("", s1, s2, s3);

Console.WriteLine("\*\*\*Concatenation\*\*\*\n" +

$"\"+\" operation : {concat\_plus}\n" +

$"method Concat : {concat\_method}\n" +

$"method Join : {concat\_join}\n");

string copy\_method = String.Copy(s1);

string assignment = s1; //приравнивание, не уверена, что так можно копировать !!!!

string sub\_method = s1.Substring(0, s1.Length);

Console.WriteLine($"\*\*\*Coping\*\*\*\n" +

$"method Copy : {copy\_method}\n" +

$"assignment : {assignment}\n" +

$"method Substring : {sub\_method}\n");

string substr1 = s3.Substring(s3.Length - 1, 1);

string substr2 = s1.Substring(7, 3);

Console.WriteLine($"\*\*\*Substring\*\*\*\n" +

$"method Substring with 1 parameter : {substr1}\n" +

$"method Substring with 2 parameters : {substr2}\n");

string[] words\_arr = concat\_plus.Split(" ");

Console.WriteLine($"\*\*\*Split\*\*\*");

foreach (string wo in words\_arr)

{

Console.WriteLine(wo);

}

string insert\_method = concat\_plus.Insert(1, "...");

Console.WriteLine($"\n\*\*\*Substring in position\*\*\*\n" +

$"method Insert : {insert\_method}\n");

string rm\_method1 = s1.Remove(s1.Length - 2);

string rm\_method2 = rm\_method1.Remove(0, 7);

Console.WriteLine($"\*\*\*Remove substring\*\*\*\n" +

$"method Remove with 1 parameter : {rm\_method1}\n" +

$"method Remove with 2 parameters : {rm\_method2}\n");

/\*Специальный знак $ идентифицирует строковый литерал как интерполированную строку.

\*Интерполированная строка — это строковый литерал, который может содержать выражения интерполяции.

\*При разрешении интерполированной строки в результирующую элементы с выражениями интерполяции заменяются

\*строковыми представлениями результатов выражений. Эта функция доступна начиная с C# 6.\*/

string interp = $"s1 : {s1}, s2 : {s2}, s3 : {s3}";

Console.WriteLine($"\*\*\*String interpolation\*\*\*\n {interp}");

//``````````````````` ЗАДАНИЕ 2 (c) ```````````````````

string n\_str = null;

string emp\_str = "";

string str = "string with text";

Console.WriteLine($"IsNullOrEmpty method\n" +

$"string is null : {String.IsNullOrEmpty(n\_str)}\n" +

$"string is empty : {String.IsNullOrEmpty(emp\_str)}\n" +

$"string with text : {String.IsNullOrEmpty(str)}\n");

Console.WriteLine($"Plus : {n\_str + emp\_str}"); //""

Console.WriteLine($"Compare : {String.Compare(n\_str, emp\_str)}"); //-1

Console.WriteLine($"String.Equals : {String.Equals(n\_str, emp\_str)}"); //False

Console.WriteLine($"String.Concat : {String.Concat(n\_str, emp\_str)}"); //""

Console.WriteLine($"Interpolation : {n\_str}, {emp\_str}"); //, ""

//``````````````````` ЗАДАНИЕ 2 (d) ```````````````````

StringBuilder str2d = new StringBuilder("Can you please sit the fuck down"); //выделяет больше памяти, чем нужно в данный момент, для возможности менять строку

Console.WriteLine($"Remove : {str2d.Remove(0, 23)}");

str2d.Insert(0, "->");

str2d.Append("<-");

Console.WriteLine($"Insert + Append: {str2d}");

//------------------------------------------------------------------------------------------------------------------------------------

//``````````````````` ЗАДАНИЕ 3 (a) ```````````````````

int[,] nums = { { 0, 1, 2, 3 }, { 4, 5, 6, 7 }, { 8, 9, 10, 11 } };

int rows = nums.GetUpperBound(0) + 1;

int cols = nums.GetUpperBound(1) + 1;

for (int i = 0; i < rows; i++)

{

for (int j = 0; j < cols; j++)

{

Console.Write(nums[i, j] + "\t");

}

Console.WriteLine("\n");

}

//``````````````````` ЗАДАНИЕ 3 (b) ```````````````````

string[] strs = new string[6] { "I was born on the wrong side of the train tracks",

"I was raised with a strap across my back",

"Lay me on my side or hold me up to the light, yeah",

"I was burned by the cold kiss of a vampire",

"I was bit by the whisper of a soft liar",

"Any good friend of yours is a good friend of mine"};

Console.WriteLine("\*\*\*Array\*\*\*");

foreach (string s in strs)

{

Console.WriteLine(s);

}

Console.WriteLine($"\nArray's length : {strs.Length}\n");

Console.Write("Введите номер строки : ");

int ind = Convert.ToInt32(Console.ReadLine()) - 1;

//если введенное число больше, количсетва строк в массиве, то менять будем последнюю строку

if (ind >= strs.Length)

{

ind = strs.Length - 1;

}

Console.Write("Введите, на что заменить эту строку : ");

string new\_str = Console.ReadLine();

strs[ind] = new\_str;

Console.WriteLine("\n\*\*\*Array witn changed line\*\*\*");

foreach (string s in strs)

{

Console.WriteLine(s);

}

//``````````````````` ЗАДАНИЕ 3 (c) ```````````````````

float[][] arr = new float[3][];

arr[0] = new float[2];

arr[1] = new float[3];

arr[2] = new float[4];

//ВВОД МАССИВА

Console.Write("Введите в массив 9 вещественных чисел через пробел : ");

//через пробел делим строку на элементы массива и записываем получившиеся строки в массив

string[] usual\_arr\_str = (Console.ReadLine().Split(" "));

int usual\_ind = 0;

for (int i = 0; i < 3; i++)

{

int max\_col = 0;

if (i == 0)

{

max\_col = 2;

}

else if (i == 1)

{

max\_col = 3;

}

else if (i == 2)

{

max\_col = 4;

}

for (int j = 0; j < max\_col; j++)

{

//элементы массива строк приводим к вещественным числам

arr[i][j] = Convert.ToSingle(usual\_arr\_str[usual\_ind]);

usual\_ind++;

}

}

//ВЫВОД МАССИВА

for (int i = 0; i < 3; i++)

{

Console.Write("\n");

int max\_col = 0;

if (i == 0)

{

max\_col = 2;

}

else if (i == 1)

{

max\_col = 3;

}

else if (i == 2)

{

max\_col = 4;

}

for (int j = 0; j < max\_col; j++)

{

Console.Write(arr[i][j] + "\t");

}

Console.WriteLine("\n");

}

//``````````````````` ЗАДАНИЕ 3 (d) ```````````````````

var str3d = "From Today on, It's My Turn!!";

var arr3d = new[] { 1.2, 7.09, 8.3 };

Console.WriteLine($"String : {str3d}");

Console.Write($"Array : ");

foreach (float i in arr3d)

{

Console.Write(i + "\t");

}

Console.Write("\n");

//------------------------------------------------------------------------------------------------------------------------------------

//``````````````````` ЗАДАНИЕ 4 (a, b) ```````````````````

(int, string, char, string, ulong) cor = (7, "Kyou kara Ore wa!!", 'A', "Rainforest Alliance", 45);

Console.WriteLine(cor);

Console.WriteLine($"\n1 cortege element : {cor.Item1}");

Console.WriteLine($"2 cortege element : {cor.Item2}");

Console.WriteLine($"3 cortege element : {cor.Item3}");

Console.WriteLine($"4 cortege element : {cor.Item4}");

Console.WriteLine($"5 cortege element : {cor.Item5}");

//``````````````````` ЗАДАНИЕ 4 (c) ```````````````````

//деконструирование кортежа

//явное объявление первых двух переменных и ключевое слово var, чтобы C# сам определил тип третьей переменной

(string song, int num, var fl) = ("Smells Like Teen Spirit", 10, 9.4f);

//C# объявлет тип всех переменных

var (int\_num, cha) = (-10, 'E');

//вы можете разложить кортеж на уже объявленные переменные

int a4c;

int b4c = 9;

(a4c, b4c) = (10, 7);

//``````````````````` ЗАДАНИЕ 4 (d) ```````````````````

(int, int, int, int, int) cor1 = (10, 20, 9, 0, 40);

(int, int, int, int, int) cor2 = (20, 10, 9, 75, -2);

//позволяет сравнивать кортежи с разным количеством элементов

Console.WriteLine("cor2.Equals(cor1) : " + cor2.Equals(cor1)); //False

//компилируются только, если количество элементов в первом кортеже равно количеству во втором

Console.WriteLine("cor1 == cor2 : " + (cor1 == cor2)); //False

Console.WriteLine("cor2.CompareTo(cor1) : " + cor2.CompareTo(cor1)); //1

Console.WriteLine("cor1.CompareTo(cor2) : " + cor1.CompareTo(cor2)); //-1

//------------------------------------------------------------------------------------------------------------------------------------

//``````````````````` ЗАДАНИЕ 5 ```````````````````

static (int, int, int, char) Method(int[] arr, string str)

{

int min;

int max;

int sum;

//локальная функция - это функция внутри другой функции (метода)

void func()

{

min = arr[0];

max = arr[0];

sum = 0;

foreach (int num in arr)

{

if (num > max)

{

max = num;

}

if (num < min)

{

min = num;

}

sum += num;

}

}

func();

return (max, min, sum, str[0]);

}

var result = Method(new int[] { 8, 9, 0, -1, 45 }, "hello there!");

Console.WriteLine("max, min, sum, first letter : " + result);

//------------------------------------------------------------------------------------------------------------------------------------

//``````````````````` ЗАДАНИЕ 6 ```````````````````

void ch(int a)

{

try

{

checked

{

a = a + 8;

}

Console.WriteLine(a);

}

catch (OverflowException)

{

Console.WriteLine("Переполнение");

}

}

void unch(int a)

{

unchecked

{

a = a + 8;

}

Console.WriteLine(a);

}

int max\_int = 2147483647;

ch(max\_int);

unch(max\_int);

}

}

}

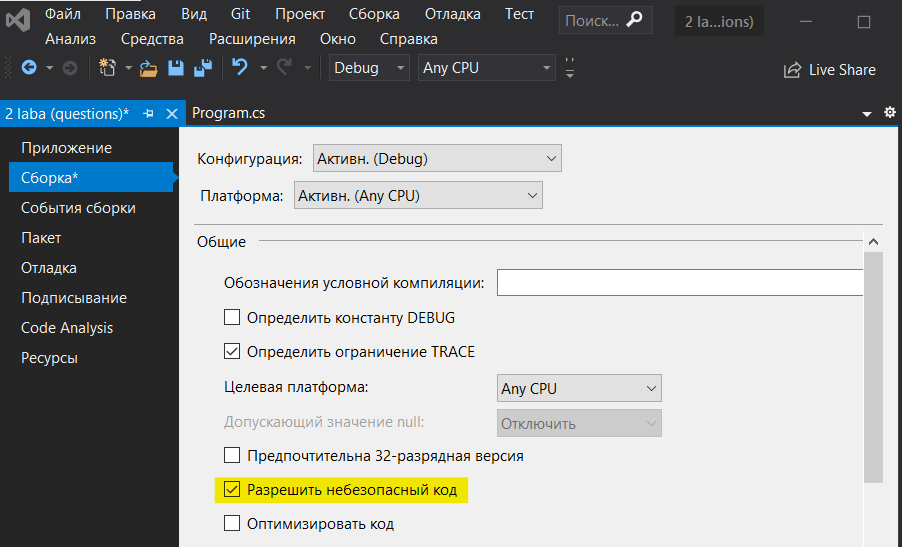
**Дополнительно**

**1) Ознакомьтесь с концепцией «небезопасного кода и указателей» в .NET. Познакомьтесь с ключевыми словами unsafe и fixed.**

Если вы программировали на С/С++, то, возможно, вы знакомы с таким понятием как **указатели**. Указатели позволяют получить доступ к определенной ячейке памяти и произвести определенные манипуляции со значением, хранящимся в этой ячейке.

В языке C# указатели очень редко используются, однако в некоторых случаях можно прибегать к ним для оптимизации приложений. Код, применяющий указатели, еще называют небезопасным кодом. Однако это не значит, что он представляет какую-то опасность. Просто при работе с ним все действия по использованию памяти, в том числе по ее очистке, ложится целиком на нас, а не на среду CLR. И с точки зрения CLR такой код не безопасен, так как среда не может проверить данный код, поэтому повышается вероятность различного рода ошибок.

Чтобы использовать небезопасный код в C#, надо первым делом указать проекту, что он будет работать с небезопасным кодом. Для этого надо установить в настройках проекта соответствующий флаг - в меню Project (Проект) найти Свойства <название проекта>. Затем в меню Build (Сборка) установить флажок **Allow unsafe code (Разрешить небезопасный код)**:



Теперь мы можем приступать к работе с небезопасным кодом и указателями.

### Ключевое слово unsafe

Блок кода или метод, в котором используются указатели, помечается ключевым словом **unsafe**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | static void Main(string[] args)  {      // блок кода, использующий указатели      unsafe      {        }  } |

Метод, использующий указатели:

|  |  |
| --- | --- |
| 1  2  3  4 | unsafe private static void PointerMethod()  {    } |

Также с помощью unsafe можно объявлять структуры:

|  |  |
| --- | --- |
| 1  2  3  4 | unsafe struct State  {    } |

### Операции \* и &

Ключевой при работе с указателями является операция \*, которую еще называют операцией разыменовывания. Операция разыменовывания позволяет получить или установить значение по адресу, на который указывает указатель. Для получения адреса переменной применяется операция &:

static void Main(string[] args)

{

    unsafe

    {

        int\* x; // определение указателя

        int y = 10; // определяем переменную

        x = &y; // указатель x теперь указывает на адрес переменной y

        Console.WriteLine(\*x); // 10

        y = y + 20;

         Console.WriteLine(\*x);// 30

        \*x = 50;

        Console.WriteLine(y); // переменная y=50

    }

    Console.ReadLine();

}

При объявлении указателя указываем тип int\* x; - в данном случае объявляется указатель на целое число. Но кроме типа int можно использовать и другие: sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal или bool. Также можно объявлять указатели на типы enum, структуры и другие указатели.

Выражение x = &y; позволяет нам получить адрес переменной y и установить на него указатель x. До этого указатель x не на что не указывал.

После этого все операции с y будут влиять на значение, получаемое через указатель x и наоборот, так как они указывают на одну и ту же область в памяти.

Для получения значения, которое хранится в области памяти, на которую указывает указатель x, используется выражение \*x.

### Получение адреса

Используя преобразование указателя к целочисленному типу, можно получить адрес памяти, на который указывает указатель:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | int\* x; // определение указателя  int y = 10; // определяем переменную    x = &y; // указатель x теперь указывает на адрес переменной y    // получим адрес переменной y  uint addr = (uint)x;  Console.WriteLine("Адрес переменной y: {0}", addr); |

Так как значение адреса - это целое число, а на 32-разрядных системах диапазон адресов 0 до 4 000 000 000, то для получения адреса используется преобразование в тип uint, long или ulong. Соответственно на 64-разрядных системах диапазон доступных адресов гораздо больше, поэтому в данном случае лучше использовать ulong, чтобы избежать ошибки переполнения.

[**Оператор fixed**]

Оператор fixed запрещает сборщику мусора (garbage collector, GC) физическое перемещение переменной в памяти, что происходит в следующей форме:

**fixed** ( type\* ptr = expr ) statement

Здесь:

**type** не обслуживаемый тип (unmanaged type) или void.  
**ptr** имя указателя.  
**expr** выражение, которое неявно конвертируется в type\*.  
**statement** выполняемый оператор или блок кода (код за фигурными скобками {}).

Оператор fixed разрешено использовать только в unsafe-контексте (код, ограниченный действием ключевого слова unsafe).

Оператор fixed устанавливает указатель на адрес managed-переменной, и одновременно "прикалывает" переменную к её физическому размещению в памяти на время выполнения оператора fixed. Без fixed указатели на managed-переменые были бы мало полезны, поскольку сборщик мусора мог бы непредсказуемо в любой момент переместить переменную в памяти.

Примечание: фактически компилятор C# не позволит Вам установить значение указателя на managed-переменную, если это не делается под управлением оператора fixed.

**unsafe**  
               {  
                    // создаем объект класса  
                    Book book = new Book();  
                    book.Age = 45;  
                    book.Price = 30;  
  
                    // указатель на структуру  
                    **fixed** (int\* **p** = &book.Price)  
                    {  
                         // указатель p указывает на адрес переменной value  
                         \***p** = 21;  
  
                         // покажем значение поля класса  
                         Console.WriteLine(book.Price);  
                         // 21  
                    }

Вы можете инициализировать указатель адресом массива или строки:

**fixed** (**int**\* p = arr) ... // эквивалентно p = &arr[0]

**fixed** (**char**\* p = str) ... // эквивалентно p = &str[0]

Вы можете инициализировать несколько указателей, пока они имеют один и тот же тип:

**fixed** (**byte**\* ps = srcarray, pd = dstarray) {...}

Чтобы инициализировать указатели разных типов, просто используйте вложение для операторов fixed:

**fixed** (**int**\* p1 = &p.x)

**fixed** (**double**\* p2 = &array[**5**])

// здесь делайте что-то с p1 и p2

Нельзя модифицировать указатели внутри операторов fixed.

После завершения действия statement (т. е. оператора или блока кода fixed) любые "привязанные" к физической памяти переменные "отвязываются", и эти переменные снова попадают под действие сборщика мусора. Таким образом, не указывайте на эти переменные вне действия оператора fixed.

**2) Ознакомьтесь с работой сборщика мусора (garbage collector, GC) в .NET.** <https://docs.microsoft.com/en-us/dotnet/standard/garbagecollection/fundamentals>

Ранее в теме [Типы значений и ссылочные типы](https://metanit.com/sharp/tutorial/2.16.php) мы рассматривали отдельные типы данных и как они располагаются в памяти. Так, при использовании переменных типов значений в методе, все значения этих переменных попадают в стек. После завершения работы метода стек очищается.

При использовании же ссылочных типов, например, объектов классов, для них также будет отводиться место в стеке, только там будет храниться не значение, а адрес на участок памяти в хипе или куче, в котором и будут находиться сами значения данного объекта. И если объект класса перестает использоваться, то при очистке стека ссылка на участок памяти также очищается, однако это не приводит к немедленной очистке самого участка памяти в куче. Впоследствии сборщик мусора (garbage collector) увидит, что на данный участок памяти больше нет ссылок, и очистит его.

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class Program  {      static void Main(string[] args)      {          Test();      }        private static void Test()      {          Country country = new Country();          country.x = 10;          country.y = 15;      }  }  class Country  {      public int x;      public int y;  } |

В методе Test создается объект Country. С помощью оператора **new** в куче для хранения объекта CLR выделяет участок памяти. А в стек добавляет адрес на этот участок памяти. В главном методе Main мы вызываем метод Test. И после того, как Test отработает, место в стеке очищается, а сборщик мусора очищает ранее выделенный под хранение объекта country участок памяти.

Сборщик мусора не запускается сразу после удаления из стека ссылки на объект, размещенный в куче. Он запускается в то время, когда среда CLR обнаружит в этом потребность, например, когда программе требуется дополнительная память.

Как правило, *объекты в куче располагаются неупорядочено, между ними могут иметься пустоты*. Куча довольно сильно фрагментирована. Поэтому после очистки памяти **в результате очередной сборки мусора оставшиеся объекты перемещаются в один непрерывный блок памяти**. Вместе с этим происходит **обновление ссылок**, чтобы они правильно указывали на новые адреса объектов.

Так же надо отметить, что для крупных объектов существует своя куча - **Large Object Heap**. В эту кучу помещаются объекты, размер которых больше 85 000 байт. Особенность этой кучи состоит в том, что при сборке мусора сжатие памяти не проводится по причине больших издержек, связанных с размером объектов.

Несмотря на то что, на сжатие занятого пространства требуется время, да и приложение не сможет продолжать работу, пока не отработает сборщик мусора, однако благодаря подобному подходу также происходит оптимизация приложения. Теперь чтобы найти свободное место в куче среде CLR не надо искать островки пустого пространства среди занятых блоков. Ей достаточно обратиться к указателю кучи, который указывает на свободный участок памяти, что уменьшает количество обращений к памяти.

Кроме того, чтобы снизить издержки от работы сборщика мусора, все объекты в куче разделяются по поколениям. Всего существует три поколения объектов: 0, 1 и 2-е.

К поколению 0 относятся новые объекты, которые еще ни разу не подвергались сборке мусора. К поколению 1 относятся объекты, которые пережили одну сборку, а к поколению 2 - объекты, прошедшие более одной сборки мусора.

Когда сборщик мусора приступает к работе, он сначала анализирует объекты из поколению 0. Те объекты, которые остаются актуальными после очистки, повышаются до поколения 1.

Если после обработки объектов поколения 0 все еще необходима дополнительная память, то сборщик мусора приступает к объектам из поколения 1. Те объекты, на которые уже нет ссылок, уничтожаются, а те, которые по-прежнему актуальны, повышаются до поколения 2.

Поскольку объекты из поколения 0 являются более молодыми и нередко находятся в адресном пространстве памяти рядом друг с другом, то их удаление проходит с наименьшими издержками.

### Класс System.GC

В библиотеках базовых классов доступен класс по имени **System.GC**, который позволяет программно *взаимодействовать со сборщиком мусора* за счет обращения к его статическим членам. Необходимость в непосредственном использовании этого класса в разрабатываемом коде возникает крайне редко (а то и вообще никогда). Обычно единственным случаем, когда нужно применять члены System.GC. является создание классов, предусматривающих использование на внутреннем уровне неуправляемых ресурсов. Это может быть, например, класс, работающий с основанным на С интерфейсом Windows API за счет применения протокола вызовов платформы .NET, или какая-то низкоуровневая и сложная логика взаимодействия с СОМ.

<https://metanit.com/sharp/tutorial/8.1.php>

<http://sergeyteplyakov.blogspot.com/2012/10/net.html>

<https://habr.com/ru/post/463213/>

**3) Ознакомьтесь с конструкцией using**

Ключевое слово using имеет три основных применения:

* [Инструкция using](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/using-statement) определяет область, по завершении которой объект удаляется.
* [Директива using](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/using-directive) создает псевдоним для пространства имен или импортирует типы, определенные в других пространствах имен.

# 1. Оператор using предоставляет удобный синтаксис, обеспечивающий правильное использование объектов [IDisposable](https://docs.microsoft.com/ru-ru/dotnet/api/system.idisposable).

Интерфейс IDisposable содержит один метод .Dispose(), который используется для освобождения ресурсов, которые захватил объект. При использовании Using не обязательно явно вызывать .Dispose() для объекта.

**using** (SqlConnection conn = **new** SqlConnection()) {   
// какая-нибудь SQL операция   
}

При этом компилятор генерирует следующий код:

SqlConnection conn = **new** SqlConnection();   
**try** {   
   
} **finally** {   
    // здесь для conn вызывается .Dispose()   
}

Using блоки делают код более читабельным и компактным.

В следующем примере показано использование оператора using.

string manyLines=@"This is line one

This is line two

Here is line three

The penultimate line is line four

This is the final, fifth line.";

using (var reader = new StringReader(manyLines))

{

string? item;

do {

item = reader.ReadLine();

Console.WriteLine(item);

} while(item != null);

}

### Интерфейс IDisposable

Интерфейс IDisposable объявляет один единственный метод Dispose, в котором при реализации интерфейса в классе должно происходить освобождение неуправляемых ресурсов. Например:

class Program

{

    static void Main(string[] args)

    {

        Test();

        Console.ReadLine();

    }

    private static void Test()

    {

        Person p=null;

        try

        {

            p = new Person();

        }

        finally

        {

            if (p != null)

            {

                p.Dispose();

            }

        }

    }

}

public class Person : IDisposable

{

    public string Name { get; set; }

    public void Dispose()

    {

        Console.Beep();

        Console.WriteLine("Disposed");

    }

}

В данном коде используется конструкция try...finally. По сути эта конструкция по функционалу в общем эквивалентна следующим двум строкам кода:

|  |  |
| --- | --- |
| 1  2 | Person p = new Person();  p.Dispose(); |

Но конструкцию try...finally предпочтительнее использовать при вызове метода Dispose, так как она гарантирует, что даже в случае возникновения исключения произойдет освобождение ресурсов в методе Dispose.

**2.** вопрос 13

<https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/using-statement>

<https://metanit.com/sharp/tutorial/8.2.php>

**Вопросы**

**1. Что такое .Net Framework и из чего он состоит?**

.NET долгое время развивался премущественно как платформа для Windows под названием .NET Framework. В 2019 вышла последняя версия этой платформы - .NET Framework 4.8. Она больше не развивается

С 2014 Microsoft стал развивать альтернативную платформу - .NET Core, которая уже предназначалась для разных платформ и должна была вобрать в себя все возможности устаревшего .NET Framework и добавить новую функциональность. Затем Microsoft последовательно выпустил ряд версий этой платформы: .NET Core 1, .NET Core 2, .NET Core 3. Логическим развитием .NET Core 3.0 стала расматриваемая в этом руководстве платформа .NET 5. Поэтому следует различать .NET Framework, который предназначен преимущественно для Windows, и кросплатформенный .NET 5. В данном руководстве речь будет идти о C# в связке с .NET 5, поскольку это актуальная платформа.

##### **1. Что являет собой среда .NET Framework?**

Платформа **.NET Framework** — это технология, которая поддерживает создание и выполнение веб-служб и приложений Windows.

.NET Framework служит средой для поддержки, разработки и выполнения распределенных приложений, которые базируются на компонентах (элементах управления).

Приложения (программы) можно разрабатывать на разных языках программирования, которые поддерживают эту технологию.

.NET Framework обеспечивает:

* совместное использование разных языков программирования;
* безопасность и переносимость программ;
* общую модель программирования на базе платформы Windows.

##### **2. Какие основные составляющие .NET Framework?**

С точки зрения программирования, .NET Framework состоит из двух основных составляющих:

* общеязыковая исполнительная среда CLR (Common Language Runtime);
* библиотека базовых классов.

Основой платформы .NET Framework является среда **CLR**. Среду выполнения можно считать агентом, который *управляет* кодом во время выполнения и предоставляет основные службы, такие как управление памятью, управление потоками и удаленное взаимодействие. При этом средой накладываются условия строгой типизации и другие виды проверки точности кода, обеспечивающие безопасность и надежность. Фактически основной задачей среды выполнения является управление кодом. Код, который обращается к среде выполнения, называют управляемым кодом, а код, который не обращается к среде выполнения, называют неуправляемым кодом. **Библиотека классов** является комплексной объектно-ориентированной коллекцией повторно используемых типов, которые применяются для разработки приложений — начиная с обычных приложений, запускаемых из командной строки, и приложений с графическим интерфейсом (GUI) и заканчивая приложениями, использующими последние технологические возможности ASP.NET, такие как веб-формы и веб-службы XML.

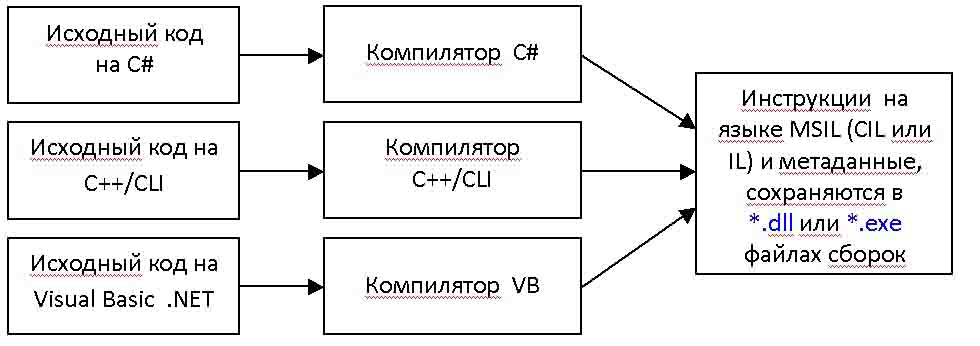
**2. Что такое CLR, FCL/BCL, CLI, IL?**

► **CLR (Common Language Runtime)** — общеязыковая исполняющая среда. Первостепенной задачей данной среды является загрузка и управления объектами .NET (чтобы программист не делал этого вручную). Общеязыковая среда выполнения CLR управляет выполнением кода .NET.

После компиляции программы на C# (или другом языке) создается файл, который содержит особого рода псевдокод или байт-код (а не исполнительный файл, как было раньше). Этот псевдокод называется Microsoft Intermediate Language (MSIL) или Common Intermediate Language (CIL). Этот псевдокод есть промежуточным языком Microsoft.

Основное назначение CLR – превратить промежуточный код MSIL в исполнительный код в процессе выполнения программы.

Любая программа, которая скомпилирована в псевдокод MSIL, может быть выполнена в любой среде, которая содержит реализацию CLR. Это обеспечивает переносность программ в среде .NET Framework.



После этого, псевдокод превращается в исполняемый код. Это осуществляет JIT -компилятор. JIT (Just-in-time) -компиляция – это компиляция на «лету».

► Объектные классы .NET, доступные для всех поддерживаемых языков программирования, содержатся в библиотеке [**Framework Class Library (FCL)**](https://ru.wikipedia.org/wiki/Framework_Class_Library)**/ Base Class Library(BCL).** Программы, написанные на любом из языков, поддерживающих платформу .NET, могут пользоваться классами и методами BCL — создавать объекты классов, вызывать их методы, наследовать необходимые классы BCL и т. д.

► **CLI** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Common Language Infrastructure*) — спецификация общеязыковой инфраструктуры. Наиболее известными реализациями этого стандарта являются [Microsoft](https://ru.wikipedia.org/wiki/Microsoft) [.NET Framework](https://ru.wikipedia.org/wiki/.NET_Framework), [Mono](https://ru.wikipedia.org/wiki/Mono), [DotGNU](https://ru.wikipedia.org/wiki/DotGNU) [Portable.NET](https://ru.wikipedia.org/wiki/Portable.NET). Спецификация CLI определяет, в частности, архитектуру исполнительной системы .NET — [CLR](https://ru.wikipedia.org/wiki/CLR) и сервисы, предоставляемые CLR выполняемым программам, классы, предоставляемые библиотекой [BCL](https://ru.wikipedia.org/wiki/Base_Class_Library), синтаксис и мнемонику общего промежуточного языка ([CIL](https://ru.wikipedia.org/wiki/MSIL)).

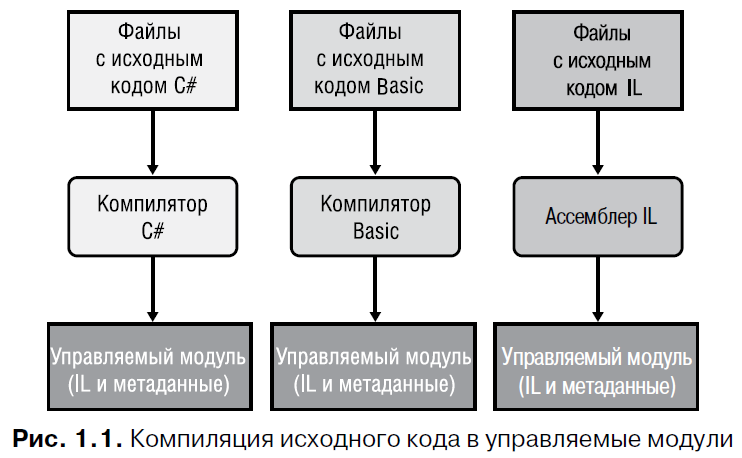
► Сперва промежуточный язык псевдокода назывался Microsoft Intermediate Language (MSIL). Позднее (в последних версиях .NET) это название было изменено на Common Intermediate Language (CIL — общий промежуточный язык). Аббревиатуры MSIL, CIL и IL (Intermediate Language)означают одно и то же.

Перед тем как код сможет выполняться CLR, любой исходный текст (на C# или другом языке) должен быть скомпилирован. Компиляция в .NET состоит из двух шагов:

1. Компиляция исходного кода в Microsoft Intermediate Language (IL)

2. Компиляция IL в специфичный для платформы код с помощью CLR

Компиляторы машинного кода производят код, ориентированный на конкретную процессорную архитектуру, например x86, х64 или ARM. В отличие от этого, все CLR-совместимые компиляторы генерируют **IL-код**. IL-код иногда называют *управляемым* (managed code), потому что CLR управляет его выполнением.

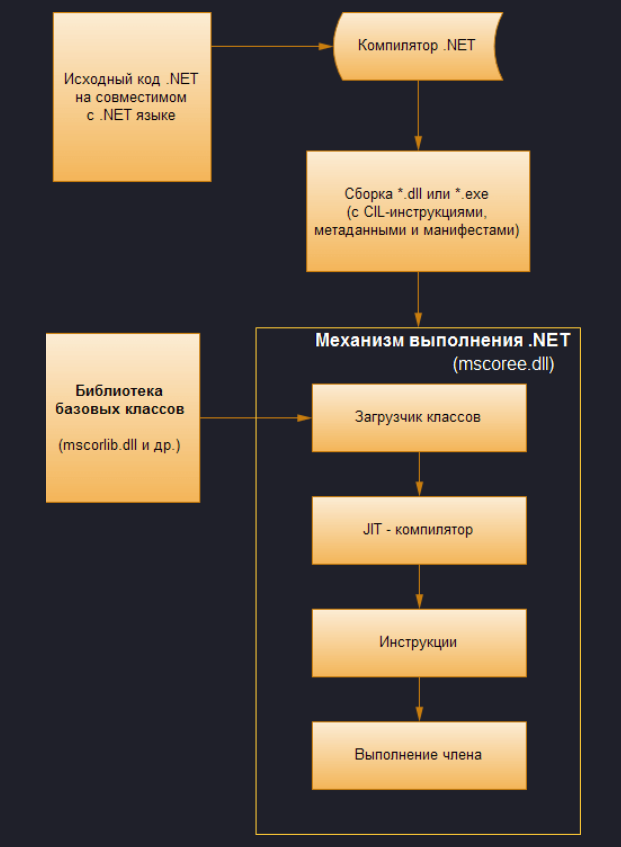


Все [компиляторы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80), поддерживающие платформу [.NET](https://ru.wikipedia.org/wiki/Microsoft_.NET), должны [транслировать](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%82%D0%BE%D1%80) код с [языков высокого уровня платформы .NET](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_.NET-%D1%8F%D0%B7%D1%8B%D0%BA%D0%BE%D0%B2) на язык CIL. В частности, код на языке CIL генерируют все компиляторы .NET фирмы [Microsoft](https://ru.wikipedia.org/wiki/Microsoft), входящие в [среду разработки](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8) [Microsoft Visual Studio](https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio) ([C#](https://ru.wikipedia.org/wiki/C_Sharp), [Managed C++](https://ru.wikipedia.org/wiki/Managed_C_Plus_Plus), [Visual Basic .NET](https://ru.wikipedia.org/wiki/Visual_Basic_.NET), [Visual J# .NET](https://ru.wikipedia.org/wiki/Visual_J_Sharp_.NET)).

По синтаксису и мнемонике язык CIL напоминает [язык ассемблера](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%B0%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80%D0%B0). Его можно рассматривать как ассемблер виртуальной машины .NET. В то же время язык CIL содержит некоторые достаточно высокоуровневые конструкции, повышающие его уровень по сравнению с ассемблером для любой реально существующей машины, и писать код непосредственно на CIL легче, чем на ассемблере для реальных машин. Поэтому CIL можно рассматривать как своеобразный «высокоуровневый ассемблер».

Язык CIL также нередко называют просто IL от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) ***i****ntermediate****l****anguage* — буквально «промежуточный язык».

Синтаксис и мнемоника языка CIL описываются стандартом «ECMA-335». Спецификация CIL является составной частью более общей спецификации — спецификации [CLI](https://ru.wikipedia.org/wiki/Common_Language_Infrastructure) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) ***c****ommon****l****anguage****i****nfrastructure*).



**3. Пояснить работу JIT-компилятора?**

JIT-компиляция (just-in-time) – процесс компиляции CIL-инструкций в соответствующий машинный код, при котором JIT-компилятор будет помещать результаты в кэш в соответствии с тем, как того требует целевая операционная система.

**4. Что такое CTS (Common Type System)?**

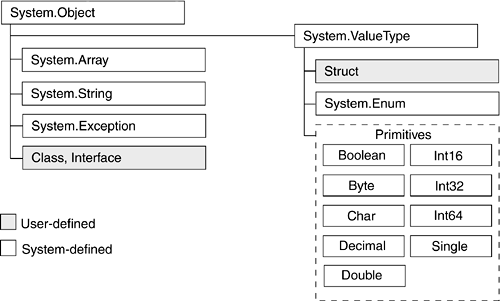
**CTS (Common Type System)** — общая система типов. Здесь описаны все возможные типы данных, поддерживаемые исполняемой средой (CLR). Нужно знать, что отдельно взятый язык, может не поддерживать абсолютно все средства, определенные спецификацией CTS. Поэтому существует:  
**CLS (Common Language Specification)**. Здесь описано подмножество общих типов и программных конструкций, которые обязательно должны поддерживать всеми языками программирования для платформы .NET Framework.

**5. Какие аспекты поведения определяет тип System.Object?**

# Системное пространство имен (System Namespace) содержит фундаментальные классы и базовые классы, которые определяют часто используемые значения и ссылочные типы данных, события и обработчики событий, интерфейсы, атрибуты и исключения обработки.

В C# предусмотрен специальный **класс object**, который неявно считается базовым классом для всех остальных классов и типов, включая и типы значений. Иными словами, все остальные типы являются производными от object. Это, в частности, означает, что переменная ссылочного типа object может ссылаться на объект любого другого типа. Кроме того, переменная типа object может ссылаться на любой массив, поскольку в C# массивы реализуются как объекты. Формально имя object считается в C# еще одним обозначением класса System.Object, входящего в библиотеку классов для среды .NET Framework.

Практическое значение этого в том, что помимо методов и свойств, которые вы определяете, также появляется доступ к множеству общедоступных и защищенных методов-членов, которые определены в классе Object. Эти методы присутствуют во всех определяемых классах.



В CLR каждый объект (а в C# всё является объектом) прямо или косвенно является производным от System.Object

//Тип, неявно производный от Object

classStudent

{

//...

}

//Тип,явно производный от Object

classPerson: System.Object

{

//...

}

**6. Что находится в mscorlib dll?**

Помимо загрузки пользовательских сборок и создания пользовательских типов, механизм CLR при необходимости будет взаимодействовать и с типами, содержащимися в библиотеках базовых классов .NET. Хотя вся библиотека базовых классов поделена на ряд отдельных сборок, главной среди них является сборка **mscorlib.dll**. В этой сборке содержится большое количество базовых типов, охватывающих широкий спектр типичных задач программирования, а также базовых типов данных, применяемых во всех языках .NET. При построении .NET-решений доступ к этой конкретной сборке будет предоставляться автоматически.

**7. Что такое «сборка»? Из чего состоит сборка .NET?**

Сборки – это файлы с расширениями \*.dll или \*.exe, которые содержат независимые от платформы .NET инструкции на промежуточном языке (Intermediate Language – IL), а также метаданные типов.

Сборка создается с помощью .NET компилятора. Сборка – это большой двоичный объект.

Сборка предназначена для сохранения пространств имен (namespaces). Пространства имен содержат типы. Типами могут быть классы, делегаты, интерфейсы, перечисления, структуры.

Сборка представляет собой коллекцию типов и ресурсов, собранных для совместной работы и образующих логическую функциональную единицу. Сборки создаются в виде исполняемого файла (EXE) или файла библиотеки динамической компоновки (DLL) и являются стандартными блоками приложений .NET. Они предоставляют сведения для среды CLR, которые нужны для распознавания реализаций типов.

В общем случае статическая сборка может состоять из четырех элементов.

* [Манифест сборки](https://docs.microsoft.com/ru-ru/dotnet/standard/assembly/manifest), который содержит метаданные сборки.
* Метаданные типов.
* Реализующий типы код на промежуточном языке IL. Он создается компилятором из одного или нескольких файлов исходного кода.
* Набор [ресурсов](https://docs.microsoft.com/ru-ru/dotnet/framework/resources/).

**8. Какие виды сборок существуют?**

Существует два вида сборок:

* однофайловые сборки;
* многофайловые сборки.

Сборка, которая состоит из одного единого модуля (\*.dll или \*.exe) называется однофайловой. В однофайловых сборках все необходимые CIL-инструкции, метаданные и манифесты размещаются в одном, четко определенном пакете.

Сборка, которая состоит из многих файлов двоичного кода .NET, называется многофайловой. Каждый из этих файлов называется модулем.

В многофайловой сборке один из модулей является главным (primary).

**9. Что такое assembly manifest?**

*Манифест* – это описание самой сборки с помощью метаданных.

В манифесте размещается информация:

* о текущей версии сборки;
* сведения о культуре (локализация строчных и графических ресурсов);
* перечень ссылок на все внешние сборки, которые нужны для правильного функционирования.

**10. Что такое GAC?**

Каждый компьютер, на котором установлена Common Language Runtime, имеет кэш кода в масштабе всей машины, называемый **Global Assembly Cache**. В глобальном кэше сборок хранятся сборки, специально предназначенные для совместного использования несколькими приложениями на компьютере.

**11. Чем managed code отличается от unmanaged code**

Нередко приложение, созданное на C#, называют **управляемым кодом** (managed code). Что это значит? А это значит, что данное приложение создано на основе платформы .NET и поэтому управляется общеязыковой средой CLR, которая загружает приложение и при необходимости очищает память. Но есть также приложения, например, созданные на языке С++, которые компилируются не в общий язык CIL, как C# или F#, а в обычный машинный код. В этом случае .NET не управляет приложением.

В то же время платформа .NET предоставляет возможности для взаимодействия с неуправляемым кодом.

**12. Как и для чего определен метод Main?**

Точкой входа в программу на языке C# является метод Main. При создании проекта консольного приложения в Visual Studio, например, создается следующий метод Main:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Program  {      static void Main(string[] args)      {          // здесь помещаются выполняемые инструкции      }  } |

По умолчанию метод Main размещается в классе Program. Название класса может быть любым. Но метод Main является обязательной частью консольного приложения. Если мы изменим его название, то программа не скомпилируется.

По сути и класс, и метод представляют своего рода блок кода: блок метода помещается в блок класса. Внутри блока метода Main располагаются выполняемые в программе инструкции.

**13. Варианты использования директивы using( using Directive ) в C#.**

Директива using позволяет использовать типы, определенные в пространстве имен, без указания полного пространства имен этого типа. В базовой форме директива using импортирует все типы из одного пространства имен, как показано в следующем примере:

using System.Text;

Теперь можно также создать псевдоним для пространства имен или типа, *используя директиву псевдонимов using*.

usingSomeUtil=MyCompany.Utilities.SomeUtility;//псевдоним

namespaceConsoleApplication

{

staticclassEntryPoint

{

staticvoidMain()

{

SomeUtilsu=newSomeUtil();

}

}

}

**14. Как связаны между собой сборки и пространства имен?**

► Пространство имен предназначено для объединения группы типов, которые связаны между собою с семантической точки зрения. Типы размещаются в сборках (assembly). Под типами понимаются классы, делегаты, интерфейсы, структуры, перечисления.

Примеры названий пространств имен:

**System**

**System.Data**

**System.IO**

**System.Collections**

**System.Threading.Tasks**

Например, в пространстве имен System.Data размещаются основные типы для работы с базами данных, в пространстве имен System.Collections размещаются основные типы для работы с коллекциями.

► Все определяемые классы и структуры, как правило, не существуют сами по себе, а заключаются в специальные контейнеры - пространства имен. Создаваемый по умолчанию класс Program уже находится в пространстве имен, которое обычно совпадает с названием проекта:

namespace HelloApp

{

    class Program

    {

        static void Main(string[] args)

        {

        }

    }

}

Пространство имен определяется с помощью ключевого слова **namespace**, после которого идет название. Так в данном случае полное название класса Program будет HelloApp.Program.

**15. Что такое примитивные типы данных? Перечислите их.**

(где-то в заданиях--)

**16. Что такое ссылочные типы? Какие типы относятся к ним?**

(где-то в заданиях--)

**17. Какие типы относятся к типам-значениям?**

Тип значения может относится к одному из двух следующих видов:

* [тип структуры](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/struct), который инкапсулирует данные и связанные функции;
* [тип перечисления](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/enum), который определяется набором именованных констант и представляет выбор или сочетание вариантов для выбора.

C# предоставляет следующие встроенные типы значений, которые также называются простыми типами.

* [Целочисленные типы](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/integral-numeric-types)
* [Числовые типы с плавающей запятой](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types)
* [bool](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/bool), представляющий логическое значение
* [char](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/char), представляющий символ Юникода UTF-16

**18. В чем отличие между ссылочными и значимыми типами данных?**

(где-то в заданиях--)

**19. Что такое упаковка и распаковка значимых типов?**

(где-то в заданиях--)

**20. В чем заключается разница между int и System.Int32? double и System.Double и т.д.?**

Типы данных, которые поддерживаются компилятором напрямую, называются примитивными (primitive types) или встроенными у них существуют прямые аналоги в библиотеке классов .NET Framework Class Library.

using short= System.Int16;

using ushort= System.UInt16;

using int= System.Int32;

using uint= System.UInt32;

**21. Для чего используется тип dynamic?**

Хотя C# относится к статически типизированным языкам, в последних версиях языка были добавлены некоторые динамические возможности. Так, начиная с .NET 4.0 была добавлена новая функциональность под названием DLR (Dynamic Language Runtime).

Чтобы понять значение данного нововведение, нужно осознавать разичие между языками со статической и динамической типизицией. В языках со *статической типизацией* выявление всех типов и их членов - свойств и методов происходит на этапе *компиляции*. А в *динамических языках* системе ничего не известно о свойствах и методах типов вплоть до *выполнения*.

Благодаря этой среде DLR C# может создавать динамические объекты, члены которых выявляются на этапе выполнения программы, и использовать их вместе с традиционными объектами со статической типизацией.

Ключевым моментом использования DLR в C# является применение типов **dynamic**. Это ключевое слово позволяет опустить проверку типов во время компиляции. Кроме того, объекты, объявленные как dynamic, могут в течение работы программы менять свой тип. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class Program  {      static void Main(string[] args)      {          dynamic x = 3; // здесь x - целочисленное int          Console.WriteLine(x);            x = "Привет мир"; // x - строка          Console.WriteLine(x);            x = new Person() { Name = "Tom", Age = 23 }; // x - объект Person          Console.WriteLine(x);            Console.ReadLine();      }  }  class Person  {      public string Name {get;set;}      public int Age { get; set; }        public override string ToString()      {          return Name + ", " + Age.ToString();      }  } |

Несмотря на то, что переменная x меняет тип своего значения несколько раз, данный код будет нормально работать. В этом использование типов dynamic отличается от применения ключевого слова **var**. Для переменной, объявленной с помощью ключевого слова var, тип выводится во время компиляции и затем во время выполнения больше не меняется.

Также можно найти общее между использованием dynamic и типом object. Если в предыдущем примере мы заменим dynamic на object: object x = 3;, то результат будет тот же. Однако и тут есть различия. Например:

|  |  |
| --- | --- |
| 1  2  3  4 | object obj = 24;  dynamic dyn = 24;  obj += 4; // так нельзя  dyn += 4; // а так можно |

На строке obj += 4; мы увидим ошибку, так как операция += не может быть применена к типам object и int. С переменной, объявленной как dynamic, это пройдет, так как ее тип будет известен только во время выполнения.

Еще одна отличительная особенность использования dynamic состоит в том, что это ключевое слово применяется не только к переменным, но и к свойствам и методам. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class Person  {      public string Name {get;set;}      public dynamic Age { get; set; }        // выводим зарплату в зависимости от переданного формата      public dynamic getSalary(dynamic value, string format)      {          if (format=="string")          {              return value + " рублей";          }          else if (format == "int")          {              return value;          }          else          {              return 0.0;          }      }        public override string ToString()      {          return Name + ", " + Age.ToString();      }  } |

В классе Person определено динамическое свойство Age, поэтому при задании значения этому свойству мы можем написать и person.Age=33, и person.Age="тридцать три". Оба варианта будут правильными.

Также есть метод getSalary, возвращающий значение dynamic. Например, в зависимости от параметра мы можем вернуть или строковое представление суммы дохода или численное. Также метод принимает dynamic в качестве параметра. Таким образом, мы можем передать в качестве значения дохода как целое, так и дробное число. Посмотрим на конкретное применение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | dynamic person1 = new Person() { Name = "Том", Age = 27 };  Console.WriteLine(person1);  Console.WriteLine(person1.getSalary(28.09, "int"));    dynamic person2 = new Person() { Name = "Билл", Age = "Двадцать два года" };  Console.WriteLine(person2);  Console.WriteLine(person2.getSalary(30, "string")); |

**22. В чем заключается главное отличие между var и dynamic?**

(21 вопрос)

**23. Что такое неявно типизированная переменная?**

(где-то в заданиях--)

**24. Для чего используют Nullable тип?**

(где-то в заданиях--)

**25. Как объявить строковый литерал? Какие операции можно выполнять со строкой?**

(где-то в заданиях--)

**26. Какие есть способы для задания и инициализации строк?**

Создавать строки можно, как используя переменную типа string и присваивая ей значение, так и применяя один из конструкторов класса String:

|  |  |
| --- | --- |
| 1  2  3  4  5 | string s1 = "hello";  string s2 = null;    string s3 = new String('a', 6); // результатом будет строка "aaaaaa"  string s4 = new String(new char[]{'w', 'o', 'r', 'l', 'd'}); |

Конструктор String имеет различное число версий. Так, вызов конструктора new String('a', 6) создаст строку "aaaaaa". И так как строка представляет ссылочный тип, то может хранить значение null.

**27. Какие методы есть у типа String?**

* **Compare**: сравнивает две строки с учетом текущей культуры (локали) пользователя
* **CompareOrdinal**: сравнивает две строки без учета локали
* **Contains**: определяет, содержится ли подстрока в строке
* **Concat**: соединяет строк
* **CopyTo**: копирует часть строки, начиная с определенного индекса в массив
* **EndsWith**: определяет, совпадает ли конец строки с подстрокой
* **Format**: форматирует строку
* **IndexOf**: находит индекс первого вхождения символа или подстроки в строке
* **Insert**: вставляет в строку подстроку
* **Join**: соединяет элементы массива строк
* **LastIndexOf**: находит индекс последнего вхождения символа или подстроки в строке
* **Replace**: замещает в строке символ или подстроку другим символом или подстрокой
* **Split**: разделяет одну строку на массив строк
* **Substring**: извлекает из строки подстроку, начиная с указанной позиции
* **ToLower**: переводит все символы строки в нижний регистр
* **ToUpper**: переводит все символы строки в верхний регистр
* **Trim**: удаляет начальные и конечные пробелы из строки

**28. В чем отличие пустой и null строки?**

**NULL** означает отсутствие значения (т. е. нет значения ), а **пустая строка** означает, что есть строковое значение нулевой длины.

**29. Как можно выполнить сравнение строк?**

(где-то в заданиях--)

**30. В чем отличие типов String и StringBuilder?**

(где-то в заданиях--)

**31. Поясните явные преобразования переменных с помощью команд Convert.**

(где-то в заданиях--)

**32. Как выполнить консольный ввод/вывод?**

► Если мы хотим вывести некоторую информацию на консоль, то нам надо передать ее в метод Console.WriteLine:

using System;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

string hello = "Привет мир";

Console.WriteLine(hello);

Console.WriteLine("Добро пожаловать в C#!");

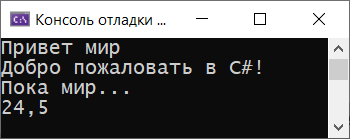
Console.WriteLine("Пока мир...");

Console.WriteLine(24.5);

}

}

}



Нередко возникает необходимость вывести на консоль в одной строке значения сразу нескольких переменных. В этом случае мы можем использовать прием, который называется интерполяцией:

string name = "Tom";

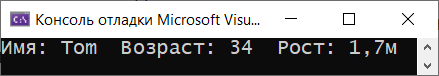
int age = 34;

double height = 1.7;

Console.WriteLine($"Имя: {name} Возраст: {age} Рост: {height}м");

Для встраивания отдельных значений в выводимую на консоль строку используются фигурные скобки, в которые заключается встраиваемое значение. Это можем значение переменной ({name}) или более сложное выражение (например, операция сложения {4 + 7}). А перед всей строкой ставится знак доллара $.

При выводе на консоль вместо помещенных в фигурные скобки выражений будут выводиться их значения:



Есть другой способ вывода на консоль сразу нескольких значений:

string name = "Tom";

            int age = 34;

            double height = 1.7;

            Console.WriteLine("Имя: {0}  Возраст: {2}  Рост: {1}м", name, height, age);

Этот способ подразумевает, что первый параметр в методе Console.WriteLine представляет выводимую строку ("Имя: {0} Возраст: {2} Рост: {1}м"). Все последующие параметры представляют значения, которые могут быть встроенны в эту строку (name, height, age). При этом важен порядок подобных параметров. Например, в данном случае вначале идет name, потом height и потом age. Поэтому у name будет представлять параметр с номером 0 (нумерация начинается с нуля), height имеет номер 1, а age - номер 2. Поэтому в строке "Имя: {0} Возраст: {2} Рост: {1}м" на место плейсхолдеров {0}, {2}, {1} будут вставляться значения соответствующих параметров.

► Кроме Console.WriteLine() можно также использовать метод **Console.Write()**, он работает точно так же за тем исключением, что не осуществляет переход на следующую строку.

►**Console.ReadLine()** позволяет получить введенную строку.

Console.Write("Введите свое имя: ");

            string name = Console.ReadLine();

            Console.WriteLine($"Привет {name}");

**33. Приведите примеры определения и инициализации одномерных и двумерных массивов.**

(где-то в заданиях--)

**34. Что такое ступенчатый массив? Как его задать?**

(где-то в заданиях--)

**35. Какие типы можно использовать в foreach? Приведите пример.**

Оператор цикла foreach предназначен для перебора элементов коллекции, строки или массива.

Общая форма оператора foreach следующая

foreach(type *identifier* in *container*)

{

// операторы

// ...

}

где

* *type* – тип переменной с именем *identifier*;
* *identifier* – имя переменной, которая используется в качестве итератора. Переменная *identifier* приобретает значение следующего элемента цикла на каждом шаге выполнения цикла foreach. Тип переменной *identifier* должен совпадать с типом массива или коллекции *container*. Связь между *identifier* и *container* реализуется с помощью союза in;
* *container*– имя коллекции или массива, который просматривается.

Оператор цикла foreach работает следующим образом. При вхождении в цикл, переменной *identifier* присваивается первый элемент массива (коллекции) *container*. На каждом следующем шаге итерации выбирается следующий элемент из *container*, который сохраняется в переменной *identifier*. Цикл завершается, когда будут пересмотрены все элементы массива (коллекции) *container*.

**36. Что такое кортеж? Для чего и как он используется?**

(где-то в заданиях--)

**37. Что такое локальная функция? Какова область ее видимости?**

(где-то в заданиях--)

**38. В чем разница между кодом, заключенным в блок checked и кодом, заключенным в блок unchecked?**

(где-то в заданиях--)

**39. Какой контекст (checked/unchecked) применяется по умолчанию? Как можно переопределить это поведение?**

По умолчанию при переполнении полученное значение усекается(unchecked контекст).

**40. Для чего используется ключевое слово fixed? Каковы особенности его использования?**

(где-то в заданиях--)