Конспекты

Оглавление

[Конспекты 1](#_Toc135079706)

[**Ассоциативный массив С++** 2](#_Toc135079707)

[Методы map 5](#_Toc135079708)

[**Коллекция Dictionary<K, V>** 8](#_Toc135079709)

[Создания и инициализация словаря 8](#_Toc135079710)

[KeyValuePair 9](#_Toc135079711)

[Перебор словаря 10](#_Toc135079712)

[Получение элементов 10](#_Toc135079713)

[Методы и свойства Dictionary 11](#_Toc135079714)

[**Регулярные выражение (regex)** 13](#_Toc135079715)

[Параметр RegexOptions 14](#_Toc135079716)

[Синтаксис регулярных выражений 15](#_Toc135079717)

[Как правильно писать регулярные выражения 19](#_Toc135079718)

[Источники: 22](#_Toc135079719)

[**WPF (XAML)** 22](#_Toc135079720)

[Пример моего кода на основе главной формы. 24](#_Toc135079721)

[Источники: 24](#_Toc135079722)

[**Конспект по C#** 25](#_Toc135079723)

[Введение 25](#_Toc135079724)

[Шаблоны проектов в С# 27](#_Toc135079725)

[Шаблон консольного приложения 27](#_Toc135079726)

[Шаблон приложения с GUI 29](#_Toc135079727)

[Среды разработки С# 31](#_Toc135079728)

[Основные типы, особенности типизации, контейнеры 34](#_Toc135079729)

[Общие сведения о классах 39](#_Toc135079730)

[Создание проекта модульного теста 44](#_Toc135079731)

[Литература 46](#_Toc135079732)

# **Ассоциативный массив С++**

Массив – контейнер данных, в котором элементы доступны по целочисленным индексам.

Ассоциативный массив – контейнер данных, в котором элементы доступны по индексам (ключам) имеющим произвольный тип

Множество – вырожденный случай ассоциативного массива, имеющий только ключи (без элементов)

**Телефонный справочник**

#include <iostream>

#include <string>

#include “phonebook.h”

using namespace std;

int

main (){

PhoneBook pBook;

pBook.addRecord (“Vasya”, “121212”);

pBook.addRecord (“Petya”, “212121”);

cout << pBook.getRecord (“Vasya”);

return 0;

}

**Использование класса map**

#include <iostream>

#include <string>

#include <map>

using namespace std;

int

main (){

map <string, string> pBook;

pBook

[“Vasya”] = “121212”;

pBook

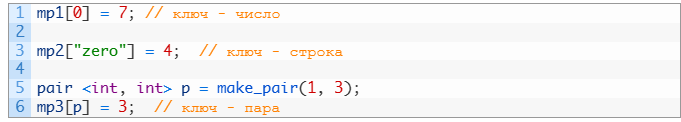
[“Petya”] = “212121”;

cout << pBook [Vasya”];

return 0;

}

В С++ ассоциативные массивы представлены в стандартной библиотеке классом std::map



**Cоздание map**

Сперва понадобится подключить соответствующую библиотеку:

|  |
| --- |
| #include <map> |

Чтобы создать map нужно воспользоваться данной конструкцией:

map < <L>, <R> > <имя>;  
<L> — этот тип данных будет относиться к значению ключа.

<R> — этот тип данных соответственно относится к значению.

map <string, int> mp; // пример

В нашем случае:

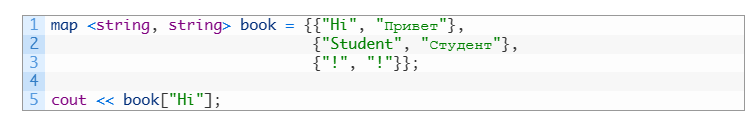
**Ключ** — строка.

**Значение** — число.

При создании map все его элементы будут иметь значение нуля.

Также имеется возможность добавить значения при инициализации (С++ 11 и выше):

**C++**



**Итераторы для map**

Использование итераторов одна из главных тем, если вам понадобится оперировать с этим контейнером. Создание итератора, как обычно происходит так:

map <тип данных> :: iterator <имя>;

* <тип данных> — <string, int> например.

С помощью его можно использовать две операции (it — итератор):

1. Чтобы обратится к ключу нужно сделать так: it->first.
2. Чтобы обратится к значению ячейки нужно сделать так: it->second.

Нельзя обращением к ключу (...->first) изменять его значение, а вот изменять таким образом значение ячейки (...->second) легко.

Нельзя использовать никакие арифметические операции над итератором:

|  |  |
| --- | --- |
| 1  2  3  4  5 | it \*= 5;    it += 3;    it /= 10; |

Все это будет считаться еще одной ошибкой для компилятора.

Для увеличения или уменьшения можно использовать инкремент или декремент.

|  |  |
| --- | --- |
| 1  2 | it++;  it--; |

Нельзя делать то же самое используя операцию присваивания (it += 1) или вычитания (it -= 1).

Чтобы не писать циклы для увеличения итератора на большие значения, можно воспользоваться функцией advance():

|  |  |
| --- | --- |
| 1  2 | advance(it, 7);  advance(it, -5); |

Она сдвигает указанный итератор вниз или вверх на указанное количество ячеек. В нашем случае он сначала увеличит на 7, а потом уменьшит на 5, в итоге получится сдвиг на две вверх.

## ****Методы**** map

Ниже мы разберем функции, которые можно использовать для работы с map.

1. ***insert***

Это функция вставки нового элемента.



* num\_1 — ключ.
* num\_2 — значение.

Мы можем сделать то же самое вот так:



1. ***count***

Возвращает количество элементов с данным ключом. В нашем случае будет возвращать — 1 или 0.

Эта функция больше подходит для multimap, у которого таких значений может быть много.



Нужно помнить, что для строк нужно добавлять кавычки — count("Good").

1. ***find***

У этой функции основная цель узнать, есть ли определенный ключ в контейнере.

* Если он есть, то передать итератор на его местоположение.
* Если его нет, то передать итератор на конец контейнера.

Например, давайте разберем данный код:



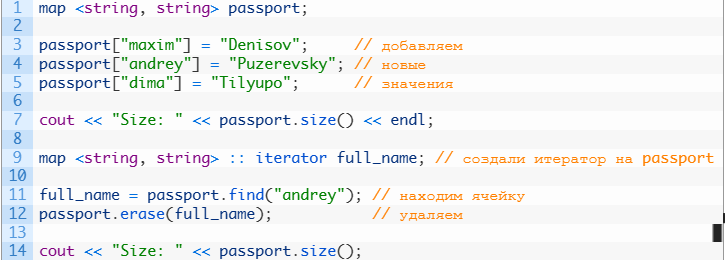
Давайте рассмотрим поподробнее:

* В строке **8**: мы создали map под названием — book.
* В строке **9**: задали единственный ключ "book" со значением "книга".
* В строке **11**: создали два итератора it и it\_2.
* В строке **13 — 14**: получаем итератор на ключ «book» и выводим его значение.
* В строке **16**: получаем итератор на ключ «books», которого нет в нашем контейнере.
* В строке **18 — 20**: проверяем не указывает ли it\_2 на конец контейнера и предупреждаем пользователя.

1. ***erase***

Иногда приходится удалять элементы. Для этого у нас есть функция — erase().

Давайте посмотрим как она работает на примере:



В итоге мы уменьшим количество наших элементов на один.

Также здесь мы воспользовались функцией size(), которая возвращает количество элементов.

passport в нашем случае хранит имя как ключ, а фамилию как значение.

Кроме того класс **map** имеет следующие функции:

* **at()** – возвращает значение по известному ключу;
* **begin()** – возвращает итератор на первый элемент контейнера;
* **clear()** – удаляет из массива все элементы;
* **empty()** – определяет, пуст ли ассоциативный массив (не содержит ни одного элемента);
* **end()** – возвращает итератор, установленный по последнему элементу контейнера;
* **max\_size()** – возвращает размер, фактически выделенный для ассоциативного контейнера с учетом его дальнейшего роста;
* **size()** – возвращает текущее количество элементов (размер) в ассоциативном контейнере;
* **swap()** – обменивает местами содержимое двух контейнеров;
* **operator=()** – присваивает один контейнер другому;
* **operator[]()** – возвращает значение (value) по заданному ключу.

**Источники:**

[Контейнеры: map, set, multiset | brestprog](https://brestprog.by/topics/containers/)

[Map в C++: что это такое и как с ним работать (codelessons.ru)](https://codelessons.ru/cplusplus/map-v-c-chto-eto-i-kak-s-etim-rabotat.html?ysclid=lemoi2h82h568173905)

[Ассоциативные массивы в C++ - GeeksforGeeks (turbopages.org)](https://translated.turbopages.org/proxy_u/en-ru.ru.4dedeafc-63fc870a-984384ef-74722d776562/https/www.geeksforgeeks.org/associative-arrays-in-cpp/)

# **Коллекция Dictionary<K, V>**

Еще один распространенный тип коллекции представляют словари. Словарь хранит объекты, которые представляют пару ключ-значение. Класс словаря Dictionary<K, V> типизируется двумя типами: параметр K представляет тип ключей, а параметр V предоставляет тип значений.

## Создания и инициализация словаря

Класс Dictionary предоставляет ряд конструкторов для создания словаря. Например, мы можем создать пустой словарь:

Dictionary<int, string> people = new Dictionary<int, string>();

Здесь словарь people в качестве ключей принимает значения типа int, а в качестве значений - строки.

При определении словаря его сразу же можно инициализировать значениями:

var people = new Dictionary<int, string>()

{

{ 5, "Tom"},

{ 3, "Sam"},

{ 11, "Bob"}

};

При инициализации применяется инициализитор - в фигурных скобках после вызова конструктора объекту передаются начальные данные. В случае со словаем мы можем передать в инициализаторе набор элементов, где каждый элемент заключается в фигурные скобки, например:

{ 5, "Tom"}

Каждый элемент представляет два значения: первое значение представляет ключ, а второе значение - собственно значение элемента. Поскольку при объявлении словаря people для ключей указан тип int, а для значений - тип string, то в элементе словаря сначала указывается число int, а затем строка. То есть в случае выше элемент имеет ключ 5, а значение - "Tom". Затем по ключу элемента мы сможем получить его значение.

Также мы можем применять другой способ инициализации:

var people = new Dictionary<int, string>()

{

[5] = "Tom",

[6] = "Sam",

[7] = "Bob"

};

При таком способе инициализации в квадратных скобках указывается ключ и ему присваивается значение элемента. Но в целом этот способ инициализации будет равноценен предыдущему.

## KeyValuePair

Стоит отметить, что каждый элемент в словаре представляет структуру ***KeyValuePair<TKey, TValue>,*** где параметр ***TKey*** представляет тип ключа, а параметр ***TValue*** - тип значений элементов. Эта структура предоставляет свойства Key и Value, с помощью которых можно получить соответственно ключ и значение элемента в словаре. И одна из версий конструктора Dictionary позволяет инициализировать словарь коллекцией объектов KeyValuePair:

var mike = new KeyValuePair<int, string>(56, "Mike");

var employees = new List<KeyValuePair<int, string>>() { mike};

var people = new Dictionary<int, string>(employees);

Конструктор типа KeyValuePair принимает два параметра - ключ элемента и его значения. То есть в данном случае создается один такой элемент - mike с ключом 56 и значением "Mike". И этот элемент добавляется в список employees, которым затем инициализируется словарь.

Можно совместить оба способа инициализации:

var mike = new KeyValuePair<int, string>(56, "Mike");

var employees = new List<KeyValuePair<int, string>>() { mike };

var people = new Dictionary<int, string>(employees)

{

[5] = "Tom",

[6] = "Sam",

[7] = "Bob",

};

В данном случае в словаре people будет четыре элемента.

## Перебор словаря

Для перебора словаря можно применять цикл foreach:

var people = new Dictionary<int, string>()

{

[5] = "Tom",

[6] = "Sam",

[7] = "Bob"

};

foreach(var person in people)

{

Console.WriteLine($"key: {person.Key} value: {person.Value}");

}

При переборе каждый элемент будет помещаться в переменную, которая представляет тип KeyValuePair, соответственно с помощью свойств Key и Value мы сможем получить ключ и значение элемента. Консольный вывод программы:

key: 5 value: Tom

key: 6 value: Sam

key: 7 value: Bob

## Получение элементов

Для обращения к элементам из словаря применяется их ключ, который передается в квадратных скобках:

словарь[ключ]

Таким образом мы можем получить и изменить элементы словаря

var people = new Dictionary<int, string>()

{

[5] = "Tom",

[6] = "Sam",

[7] = "Bob",

};

// получаем элемент по ключу 6

string sam = people[6]; // Sam

Console.WriteLine(sam); // Sam

// переустанавливаем значение по ключу 6

people[6] = "Mike";

Console.WriteLine(people[6]); // Mike

// добавляем новый элемент по ключу 22

people[22] = "Eugene";

Console.WriteLine(people[22]); // Eugene

Более того, таким образом мы можем также добавить новый элемент в словарь. При установке значения по ключу, если элемент с таким ключом уже есть в словаре, то значение переустанавливается. Если же элемента с подобным ключом нет в словаре, то элемент добавляется.:

## Методы и свойства Dictionary

Среди методов класса Dictionary можно выделить следующие:

void Add(K key, V value): добавляет новый элемент в словарь

* **void Clear():** очищает словарь
* **bool ContainsKey(K key):** проверяет наличие элемента с определенным ключом и возвращает true при его наличии в словаре
* **bool ContainsValue(V value):** проверяет наличие элемента с определенным значением и возвращает true при его наличии в словаре
* **bool Remove(K key):** удаляет по ключу элемент из словаря.

Другая версия этого метода позволяет получить удленный элемент в выходной параметр: bool Remove(K key, out V value)

* **bool TryGetValue(K key, out V value):** получает из словаря элемент по ключу key. При успешном получении передает значение элемента в выходной параметр value и возвращает true
* **bool TryAdd(K key, V value):** добавляет в словарь элемент с ключом key и значением value. При успешном добавлении возвращает true

Из свойств следует отметить свойство Count, которое возвращает количество элементов в словаре.

Применение методов:

// условная телефонная книга

var phoneBook = new Dictionary<string, string>();

// добавляем элемент: ключ - номер телефона, значение - имя абонента

phoneBook.Add("+123456", "Tom");

// альтернативное добавление

// phoneBook["+123456"] = "Tom";

// Проверка наличия

var phoneExists1 = phoneBook.ContainsKey("+123456"); // true

Console.WriteLine($"+123456: {phoneExists1}");

var phoneExists2 = phoneBook.ContainsKey("+567456"); // false

Console.WriteLine($"+567456: {phoneExists2}");

var abonentExists1 = phoneBook.ContainsValue("Tom"); // true

Console.WriteLine($"Tom: {abonentExists1}");

var abonentExists2 = phoneBook.ContainsValue("Bob"); // false

Console.WriteLine($"Bob: {abonentExists2}");

// удаление элемента

phoneBook.Remove("+123456");

// проверяем количество элементов после удаления

Console.WriteLine($"Count: {phoneBook.Count}"); // Count: 0

# **Регулярные выражение (regex)**

Классы StringBuilder и String предоставляют достаточную функциональность для работы со строками. Однако .NET предлагает еще один мощный инструмент - регулярные выражения. Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

Основная функциональность регулярных выражений в .NET сосредоточена в пространстве имен System.Text.RegularExpressions. А центральным классом при работе с регулярными выражениями является класс Regex. Например, у нас есть некоторый текст и нам надо найти в нем все словоформы какого-нибудь слова. С классом Regex это сделать очень просто:

using System.Text.RegularExpressions;

string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";

Regex regex = new Regex(@"туп(\w\*)");

MatchCollection matches = regex.Matches(s);

if (matches.Count > 0)

{

foreach (Match match in matches)

Console.WriteLine(match.Value);

}

else

{

Console.WriteLine("Совпадений не найдено");

}

Здесь мы находим в искомой строке все словоформы слова "туп". В конструктор объекта Regex передается регулярное выражение для поиска. Далее мы разберем некоторые элементы синтаксиса регулярных выражений, а пока достаточно знать, что выражение туп(\w\*) обозначает, найти все слова, которые имеют корень "туп" и после которого может стоять различное количество символов. Выражение \w означает алфавитно-цифровой символ, а звездочка после выражения указывает на неопределенное их количество - их может быть один, два, три или вообще не быть.

Метод *Matches* класса *Regex* принимает строку, к которой надо применить регулярные выражения, и возвращает коллекцию найденных совпадений.

Каждый элемент такой коллекции представляет объект *Match*. Его свойство *Value* возвращает найденное совпадение.

И в данном случае мы получим следующий консольный вывод

**тупогуб**

**тупогубенький**

**тупа**

## Параметр RegexOptions

Класс *Regex* имеет ряд конструкторов, позволяющих выполнить начальную инициализацию объекта. Две версии конструкторов в качестве одного из параметров принимают перечисление *RegexOptions*. Некоторые из значений, принимаемых данным перечислением:

* ***Compiled***: при установке этого значения регулярное выражение компилируется в сборку, что обеспечивает более быстрое выполнение
* ***CultureInvariant***: при установке этого значения будут игнорироваться региональные различия
* ***IgnoreCase***: при установке этого значения будет игнорироваться регистр
* ***IgnorePatternWhitespace***: удаляет из строки пробелы и разрешает комментарии, начинающиеся со знака #
* ***Multiline***: указывает, что текст надо рассматривать в многострочном режиме. При таком режиме символы "^" и "$" совпадают, соответственно, с началом и концом любой строки, а не с началом и концом всего текста
* ***RightToLeft***: приписывает читать строку справа налево
* ***Singleline***: при данном режиме символ "." соответствует любому символу, в том числе последовательности "\n", которая осуществляет переход на следующую строку

**Например:**

Regex regex = new Regex(@"туп(\w\*)", RegexOptions.IgnoreCase);

При необходимости можно установить несколько параметров:

Regex regex = new Regex(@"туп(\w\*)", RegexOptions.Compiled | RegexOptions.IgnoreCase);

## Синтаксис регулярных выражений

Рассмотрим вкратце некоторые элементы синтаксиса регулярных выражений:

* **^:** соответствие должно начинаться в начале строки (например, выражение @"^пр\w\*" соответствует слову "привет" в строке "привет мир")
* **$:** конец строки (например, выражение @"\w\*ир$" соответствует слову "мир" в строке "привет мир", так как часть "ир" находится в самом конце)
* **.:** знак точки определяет любой одиночный символ (например, выражение "м.р" соответствует слову "мир" или "мор")
* **\*:** предыдущий символ повторяется 0 и более раз
* **+:** предыдущий символ повторяется 1 и более раз
* **?:** предыдущий символ повторяется 0 или 1 раз
* **\s**: соответствует любому пробельному символу
* **\S**: соответствует любому символу, не являющемуся пробелом
* **\w**: соответствует любому алфавитно-цифровому символу
* **\W**: соответствует любому не алфавитно-цифровому символу
* **\d**: соответствует любой десятичной цифре
* **\D** : соответствует любому символу, не являющемуся десятичной цифрой

Это только небольшая часть элементов. Более подробное описание синтаксиса регулярных выражений можно найти на msdn в статье [Элементы языка регулярных выражений — краткий справочник.](https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference)

Теперь посмотрим на некоторые примеры использования. Возьмем первый пример с скороговоркой "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа" и найдем в ней все слова, где встречается корень "губ":

string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела была тупа";

Regex regex = new Regex(@"\w\*губ\w\*");

Так как выражение **\w\*** соответствует любой последовательности алфавитно-цифровых символов любой длины, то данное выражение найдет все слова, содержащие корень "губ".

Второй простенький пример - нахождение телефонного номера в формате 111-111-1111:

string s = "456-435-2318";

Regex regex = new Regex(@"\d{3}-\d{3}-\d{4}");

Если мы точно знаем, сколько определенных символов должно быть, то мы можем явным образом указать их количество в фигурных скобках: \d{3} - то есть в данном случае три цифры.

Мы можем не только задать поиск по определенным типам символов - пробелы, цифры, но и задать конкретные символы, которые должны входить в регулярное выражение. Например, перепишем пример с номером телефона и явно укажем, какие символы там должны быть:

string s = "456-435-2318";

Regex regex = new Regex("[0-9]{3}-[0-9]{3}-[0-9]{4}");

В квадратных скобках задается диапазон символов, которые должны в данном месте встречаться. В итоге данный и предыдущий шаблоны телефонного номера будут эквивалентны.

Также можно задать диапазон для алфавитных символов: Regex regex = new Regex("[a-v]{5}"); - данное выражение будет соответствовать любому сочетанию пяти символов, в котором все символы находятся в диапазоне от a до v.

Можно также указать отдельные значения: Regex regex = new Regex(@"[2]\*-[0-9]{3}-\d{4}");. Это выражение будет соответствовать, например, такому номеру телефона "222-222-2222" (так как первые числа двойки)

С помощью операции | можно задать альтернативные символы, например:

Regex regex = new Regex(@"(2|3){3}-[0-9]{3}-\d{4}");

То есть первые три цифры могут содержать только двойки или тройки. Такой шаблон будет соответствовать, например, строкам "222-222-2222" и "323-435-2318". А вот строка "235-435-2318" уже не подпадает под шаблон, так как одной из трех первых цифр является цифра 5.

Итак, у нас такие символы, как \*, + и ряд других используются в качестве специальных символов. И возникает вопрос, а что делать, если нам надо найти, строки, где содержится точка, звездочка или какой-то другой специальный символ? В этом случае нам надо просто экранировать эти символы слешем:

Regex regex = new Regex(@"(2|3){3}\.[0-9]{3}\.\d{4}");

// этому выражению будет соответствовать строка "222.222.2222"

**Проверка на соответствие строки формату**

Нередко возникает задача проверить корректность данных, введенных пользователем. Это может быть проверка электронного адреса, номера телефона, Класс Regex предоставляет статический метод IsMatch, который позволяет проверить входную строку с шаблоном на соответствие:

using System.Text.RegularExpressions;

string pattern = @"^(?("")(""[^""]+?""@)|(([0-9a-z]((\.(?!\.))|[-!#\$%&'\\*\+/=\?\^`\{\}\|~\w])\*)(?<=[0-9a-z])@))" +

@"(?(\[)(\[(\d{1,3}\.){3}\d{1,3}\])|(([0-9a-z][-\w]\*[0-9a-z]\*\.)+[a-z0-9]{2,17}))$";

var data = new string[]

{

"tom@gmail.com",

"+12345678999",

"bob@yahoo.com",

"+13435465566",

"sam@yandex.ru",

"+43743989393"

};

Console.WriteLine("Email List");

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

{

if (Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase))

{

Console.WriteLine(data[i]);

}

}

Переменная pattern задает регулярное выражение для проверки адреса электронной почты. Данное выражение предлагает нам Microsoft на страницах msdn.

Далее в цикле мы проходим по массиву строк и определяем, какие строки соответствуют этому шаблону, то есть представляют валидный адрес электронной почты. Для проверки соответствия строки шаблону используется метод IsMatch: Regex.IsMatch(data[i], pattern, RegexOptions.IgnoreCase). Последний параметр указывает, что регистр можно игнорировать. И если строка соответствует шаблону, то метод возвращает true.

**Замена и метод Replace**

Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой:

string text = "Мама мыла раму. ";

string pattern = @"\s+";

string target = " ";

Regex regex = new Regex(pattern);

string result = regex.Replace(text, target);

Console.WriteLine(result);

Данная версия метода Replace принимает два параметра: строку с текстом, где надо выполнить замену, и сама строка замены. Так как в качестве шаблона выбрано выражение "\s+ (то есть наличие одного и более пробелов), метод Replace проходит по всему тексту и заменяет несколько подряд идущих пробелов ординарными.

Другой пример - на вход подается номер телефона в произвольном формате, и мы хотим оставить в нем только цифры:

string phoneNumber = "+1(876)-234-12-98";

string pattern = @"\D";

string target = "";

Regex regex = new Regex(pattern);

string result = regex.Replace(phoneNumber, target);

Console.WriteLine(result); // 18762341298

В данном случае шаблон "\D" представляет любой символ, который не является цифрой. Любой такой символ заменяется на пустую строку "", то есть в итоге из строки "+1(876)-234-12-98" мы получим строку "18762341298".

## Как правильно писать регулярные выражения

Прежде, чем садиться и писать, подумайте, что именно вы хотите сделать. Регулярное выражение должно начинаться с мысли "Я хочу найти/заменить/удалить то-то и то-то". Затем вам нужен исходный текст, который содержит как ПРАВИЛЬНЫЕ, так и НЕправильные данные. Затем вы открываете <https://regex101.com/>, вставляете текст и начинаете писать регулярное выражение. Этот инструмент укажет и покажет все ошибки, а также подсветит результаты поиска.

Для примера возьмём валидацию ip-адреса. Первая мысль должна быть: "Я хочу валидировать ip-адрес. А что такое ip-адрес? Из чего он состоит?". Затем нужен список валидных и невалидных адресов:

# Валидные адреса

0.0.0.0

0.1.2.3

99.99.99.99

199.199.199.199

255.255.255.255

# Невалидные адреса

01.01.01.01

.1.2.3

1.2.3.

255.0.0.256

Валидный адрес должен содержать четыре числа (байта) от 0 до 255. Если он содержит число больше 255, это уже ошибка. Если бы мы делали валидацию на каком-либо языке программирования, то можно было бы разбить выражение на четыре части и проверить каждое число отдельно. Но регулярные выражения не поддерживают проверки больше или меньше, поэтому придётся делать по-другому.

Для начала упростим задачу: будем валидировать не весь ip-адрес, а только один байт. А байт это всегда есть либо одно-, либо дву-, либо трёхзначное число. Для одно- и двузначного числа шаблон очень простой - любая цифра. А вот для трёхзначного числа первая цифра либо единица, либо двойка. Если первая цифра единица, то вторая и третья могут быть от нуля до девяти. Если же первая цифра двойка, то вторая может быть только от нуля до пяти. Если первая цифра двойка и вторая пятёрка, то третья может быть только от ноля до пяти. Давайте формализуем:

# Валидация байта

от 0 до 9 \d

от 10 до 99 [1-9]\d

от 100 до 199 1\d\d

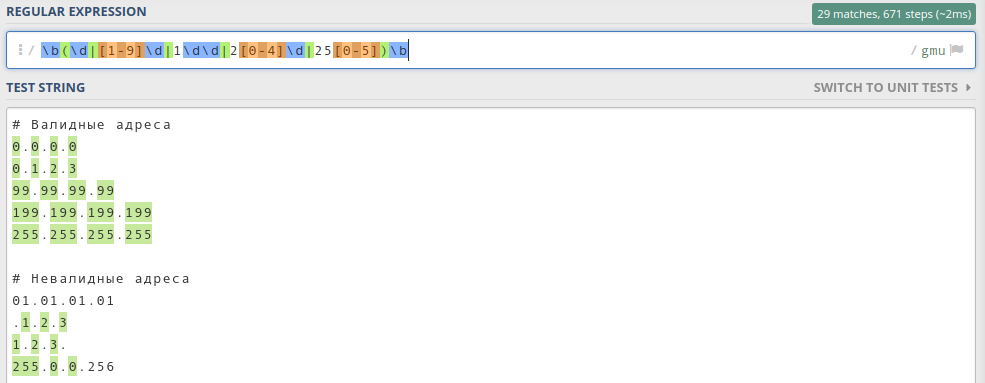
от 200 до 249 2[0-4]\d

от 250 до 255 25[0-5]

Теперь, зная все диапазоны байта, можно объединить их в одно выражение через вертикальную палочку | (ИЛИ):

\b(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\b

Пробуем регулярку в деле:

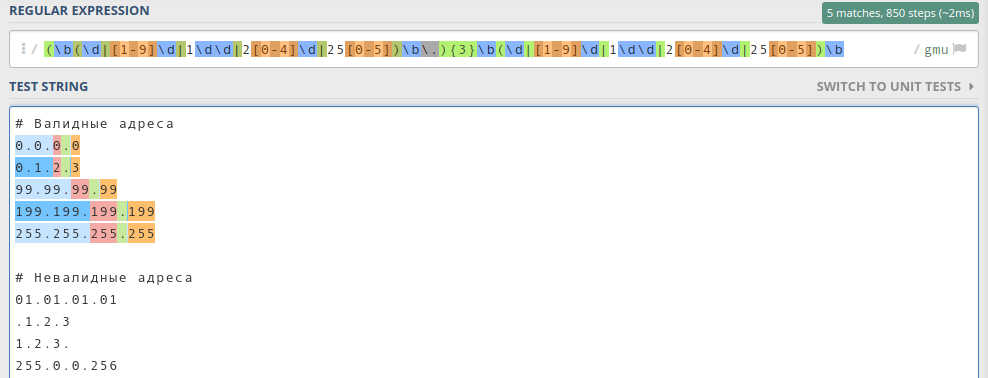


Как видим, все байты стали зелёненькими. Это значит, что мы на верном пути.

Осталось дело за малым: сделать так, чтобы искать четыре байта, а не один. Нужно учесть, что байты разделены тремя точками. То есть мы ищем три байта с точкой на конце и один без точки:

(\b(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\b\.){3}\b(\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\b

Результат выглядит так:



Подсветились только валидные ip-адреса, значит регулярное выражение работает корректно.

## Источники:

<http://microsin.net/programming/pc/csharp-before-string.html> про @

<https://metanit.com/sharp/tutorial/7.4.php>

<https://anton-pribora.ru/articles/it/regular-expressions/?ysclid=lh67k9vu3k951346125>

# **WPF (XAML)**

Визуализация в WPF тесно связана с такими понятиями как логическое и визуальное дерево. Эти деревья являются своего рода каркасом приложения. Так мы можем представить приложение как некий набор вложенных элементов. Возьмем, к примеру, следующую простейшую разметку xaml:

<Window x:Class="ControlsTemplateApp.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:ControlsTemplateApp"

mc:Ignorable="d"

Title="MainWindow" Height="250" Width="300">

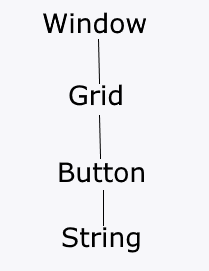
<Grid>

<Button x:Name="myButton">Hello</Button>

</Grid>

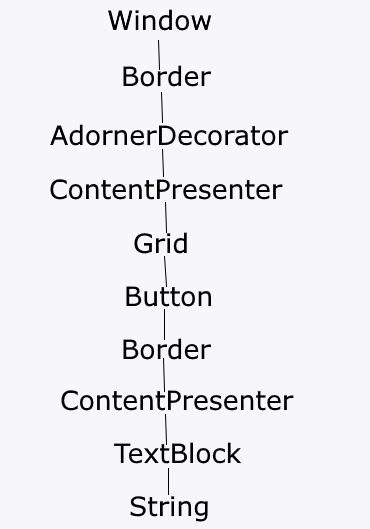
</Window>

Структуру элементов здесь можно представить следующей схемой:



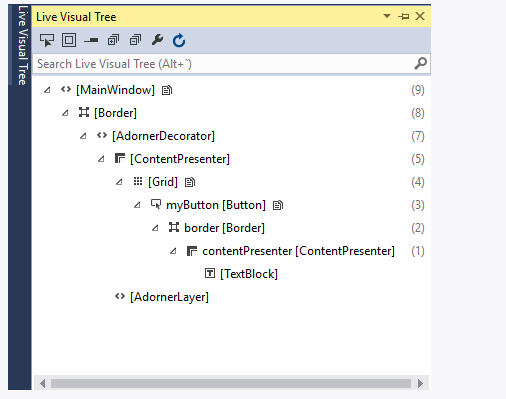
То есть в Window есть Grid, в Gride - элемент Button, в кнопке в качестве содержимого установлен некоторый текст в виде объекта String. В итоге получается некое дерево элементов, которое называется логическим. В WPF оно представлено классом System.Windows.LogicalTreeHelper. Логическое дерево имеет дело с визуализацией как таковой, оно образует модель доступа к дочерним элементам.

От него отличается визуальное дерево, представленное классом System.Windows.Media.VisualTreeHelper. Так, визуальное дерево для вышеприведенной разметки xaml будет выглядеть следующим образом:

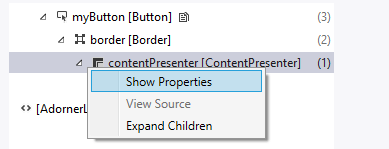


Визуальное дерево получается гораздо сложнее, оно показывает, как с визуальной точки зрения устроен элемент, из каких частей он состоит.

Visual Studio имеет встроенные средства для просмотра визуального дерева элементов. Для этого нам надо запустить проект в режиме отладки и в меню выбрать пункт Debug -> Windows -> Live Visual Tree. После нажатия на этот пункт Visual Studio откроет окно с визуальным деревом, в котором мы можем посмотреть, как устроен элемент:



Если мы нажмем правой кнопкой мыши на элемент в этом дереве и в контекстном меню выберем пункт Show Properties, справа в Visual Studio откроется окно свойств для этого элемента.



## Пример моего кода на основе главной формы.

*/// Главное дерево Window(окна)*   
<Window x:Class="MainDateBase.MainWindow"

*/// Пространство имен*

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"

xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:local="clr-namespace:MainDateBase"

mc:Ignorable="d"

*/// Title – Название, Height и Width – длина и ширина окна, Icon – икона, MaxHeight и MinHeight – ограничения максимального(минимального) размер окна*

Title="База данных библиотеки" Height="415" Width="715" Icon="/icon.ico" MaxWidth="715" MaxHeight="415" MinWidth="715" MinHeight="415">

*/// Дерево контейнера высшего уровня, в нем могу содержаться другие элементы интерфейса*

<Grid>

*/// Дерево фона в котором расписываются все его настройки*

*/// Grid.Background – Фон, в моем случае изображение*

<Grid.Background>

<ImageBrush ImageSource="/1614630795\_15-p-fon-knigi-dlya-fotoshopa-24 — копия.jpg"/>

*/// Закрытие дерева фона.*

</Grid.Background>

*/// DataGrid – элемент интерфейса(таблица), где x:Name – название элемента(изначально его нет), d:ItemsSource="{d:SampleData ItemCount=18}" – пример как будет выглядеть таблица (необязательно), Margin – его расположение на самом Grid*

<DataGrid x:Name="datagrid" d:ItemsSource="{d:SampleData ItemCount=18}" Margin="10,65,10,10"/>

*/// Button – элемент интерфейса (кнопка), где x:Name – название элемента(изначально его нет), Content – Название на самой кнопки (изначально просто Button) HorizontalAlignment и VerticalAlignment – привязки, Width – размер элемента, Click – обработчик события по нажатию на кнопку, Background – фон элемента (в данном случае голубоватый цвет).*

<Button x:Name="Button\_Save" Content="Сохранить" HorizontalAlignment="Left" Margin="115,30,0,0" VerticalAlignment="Top" Width="84" Click="Button\_Save\_Click" Background="#FFACCEF3"/>

<Button x:Name="Button\_Open" Content="Открыть" HorizontalAlignment="Left" Margin="10,30,0,0" VerticalAlignment="Top" Width="84" Click="Button\_Open\_Click" Background="#FFACCEF3"/>

<Button x:Name="Button\_Add" Content="Добавить" HorizontalAlignment="Left" Margin="221,30,0,0" VerticalAlignment="Top" Width="84" Click="Button\_Add\_Click" Background="#FFACCEF3"/>

<Button x:Name="Button\_Del" Content="Удалить строку" Margin="0,30,178,0" VerticalAlignment="Top" Click="Button\_Del\_Click" HorizontalAlignment="Right" Width="100" Background="#FFACCEF3"/>

<Button x:Name="Button\_Del\_All" Content="Удалить все" Margin="0,30,52,0" VerticalAlignment="Top" Click="Button\_Del\_All\_Click" HorizontalAlignment="Right" Width="100" Background="#FFACCEF3"/>

<Button x:Name="Button\_Sort" Content="Сортировка" HorizontalAlignment="Left" Margin="324,30,0,0" VerticalAlignment="Top" Click="Button\_Sort\_Click" Width="84" Background="#FFACCEF3"/>

*/// TextBlock – элемент интерфейса(изменяющийся лейбл), TextWrapping – способ переноса текста(по умолчание NoWrap), Text – Изначальный текст в лейбле*

<TextBlock x:Name="Label\_Save" Margin="563,4,2,0" TextWrapping="Wrap" Text="Автосохранение" VerticalAlignment="Top"/>

*/// Menu – элемент интерфейса(контейнер на подобие Grid)*

<Menu Margin="0,0,157,0" Height="30" VerticalAlignment="Top">

<Menu.Background>

<ImageBrush/>

</Menu.Background>

*/// MenuItem – Элемент меню, с выбором подменю (дерево, пример Файл – подменю открыть)*

<MenuItem Header="Файл">

<MenuItem Header="Открыть" Click="Open\_Click"/>

<MenuItem Header="Сохранить" Click="Save\_Click"/>

</MenuItem>

<MenuItem Header="Справка">

<MenuItem Header="Об авторе" Click="Author\_Click" />

</MenuItem>

*/// Закрытие дерева Menu*

</Menu>

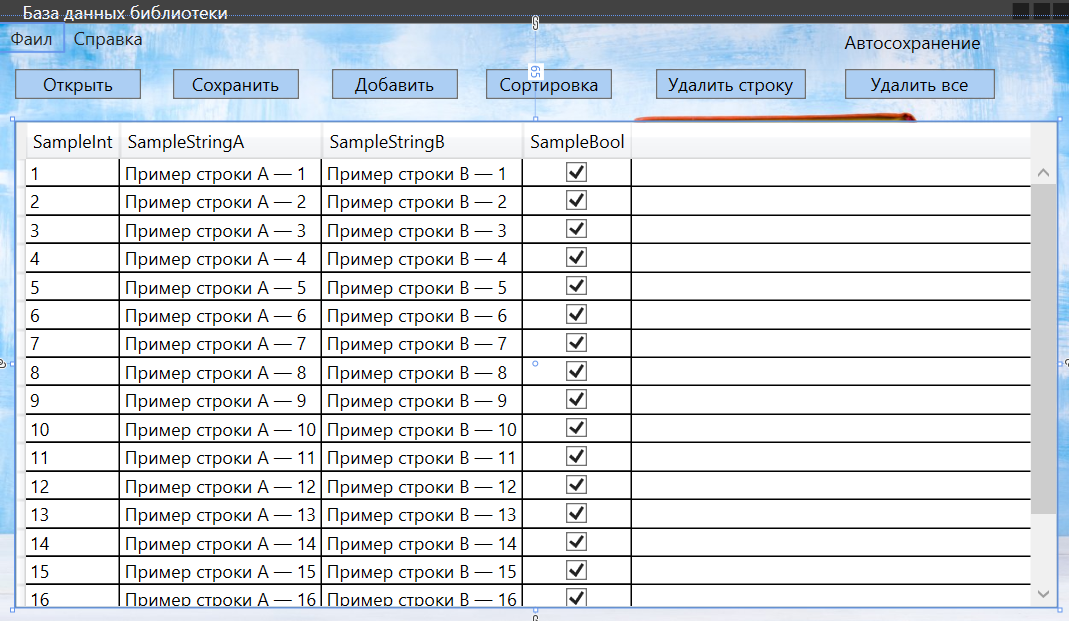
*/// Закрытие дерева Grid*

</Grid>

*/// Закрытие дерева Window*

</Window>

В результате получаем такую форму



## Источники:

<https://csharp.webdelphi.ru/pervoe-prilozhenie-wpf/>

<https://metanit.com/sharp/wpf/12.php?ysclid=lhj73nowrt520623710>

# **Конспект по C#**

Введение

C#, C-sharp, си-шарп - язык программирования, сочетающий объектно-ориентированные и аспектно-ориентированные концепции. Разработан в 1998—2001 годах группой инженеров под руководством Андерсa Хейлсбергa в компании Microsoft как основной язык разработки приложений для платформы Microsoft .NET. Компилятор с C# входит в стандартную установку самой .NET, поэтому программы на нём можно создавать и компилировать даже без инструментальных средств вроде Visual Studio.

C# относится к семье языков с C-подобным синтаксисом, из них его синтаксис наиболее близок к С++ и Java. Язык имеет строгую статическую типизацию, поддерживает полиморфизм, перегрузку операторов, указатели на функции-члены классов, атрибуты, события, свойства, исключения, комментарии в формате XML. Переняв многое от своих предшественников — языков С++, Java, Delphi, Модула и Smalltalk — С#, опираясь на практику их использования, исключает некоторые модели, зарекомендовавшие себя как проблематичные при разработке программных систем: так, C# не поддерживает множественное наследование классов (в отличие от C++).

**Особенности языка**

C# разрабатывался как язык программирования прикладного уровня для CLR и, как таковой, зависит, прежде всего, от возможностей самой CLR. Это касается, прежде всего, системы типов C#, которая отражает FCL. Присутствие или отсутствие тех или иных выразительных особенностей языка диктуется тем, может ли конкретная языковая особенность быть транслирована в соответствующие конструкции CLR. Так, с развитием CLR от версии 1.1 к 2.0 значительно обогатился и сам C#; подобного взаимодействия следует ожидать и в дальнейшем. (Однако эта закономерность была нарушена с выходом C# 3.0, представляющим собой расширения языка, не опирающиеся на расширения платформы .NET.) CLR предоставляет C#, как и всем другим .NET-ориентированным языкам, многие возможности, которых лишены «классические» языки программирования. Например, сборка мусора не реализована в самом C#, а производится CLR для программ, написанных на C# точно так же, как это делается для программ на VB.NET, J# и др.

**Название языка**

Символ # в названии языка можно интерпретировать и как две пары плюсов ++, намекающие на новый шаг в развитии языка по сравнению с C++ (подобно шагу от C к C++), и как музыкальный символ диез, вместе с буквой C составляющий в английском языке название ноты до-диез. Последнее и дало название языку. Несмотря на то, что символ # (октоторп) в действительности является символом для обозначения номера на большинстве клавиатур и отличается от символа диез # (Unicode U+266F), Microsoft, как автор языка, неоднократно обращалась к своим клиентам с просьбой принять это допущение.

**С# последняя версия стандарта**

**C# версии 11**

*Выпущено в ноябре 2022 г.*

В C# 11 были добавлены следующие функции:

* Необработанные строковые литералы
* Поддержка универсальной математики
* Универсальные атрибуты
* Строковые литералы UTF-8
* Новые строки в выражениях интерполяции строк
* Шаблоны списков
* Локальные типы файлов
* Обязательные члены
* Автоматические структуры по умолчанию
* Сопоставление Span<char> шаблонов в константе string
* Расширенная nameof область
* Числовой intPtr
* ref поля и scoped ref
* Улучшенное преобразование групп методов для делегирования.
* Предупреждающий сигнал 7

В C# 11 представлена универсальная математика и несколько функций, поддерживающих эту цель. Числовые алгоритмы можно написать один раз для всех числовых типов. Существует больше возможностей, которые упрощают работу с типами struct , например необходимые члены и автоматические структуры по умолчанию. Работа со строками упрощается благодаря необработанным строковым литералам, новой строке в интерполяциях строк и строковым литералам UTF-8. Такие функции, как локальные типы файлов, упрощают генераторы источников. Наконец, шаблоны списков добавляют дополнительную поддержку для сопоставления шаблонов.

Шаблоны проектов в С#

Чтобы получить доступ к шаблонам проектов UWP C#, когда вы создаете проект в Visual Studio, отфильтруйте язык, выбрав C# , платформу, выбрав Windows, и тип проекта, выбрав значение UWP.

Эти шаблоны проектов можно использовать для создания приложений UWP на C#.

|  |  |
| --- | --- |
| Пустое приложение (универсальные приложения для Windows) | Создает приложение UWP. Созданный проект содержит базовую страницу, производную от класса [Windows.UI.Xaml.Controls.Page](https://learn.microsoft.com/ru-ru/uwp/api/windows.ui.xaml.controls.page), с помощью которой можно приступить к созданию пользовательского интерфейса. |
| Приложение модульных тестов (универсальное приложение для Windows) | Создает проект модульного теста на C# для приложения UWP. Дополнительные сведения см. в статье [Модульное тестирование кода C#](https://learn.microsoft.com/ru-ru/visualstudio/test/unit-testing-visual-csharp-code-in-a-store-app). |

С помощью этих шаблонов проектов можно создавать фрагменты приложений UWP на C#.

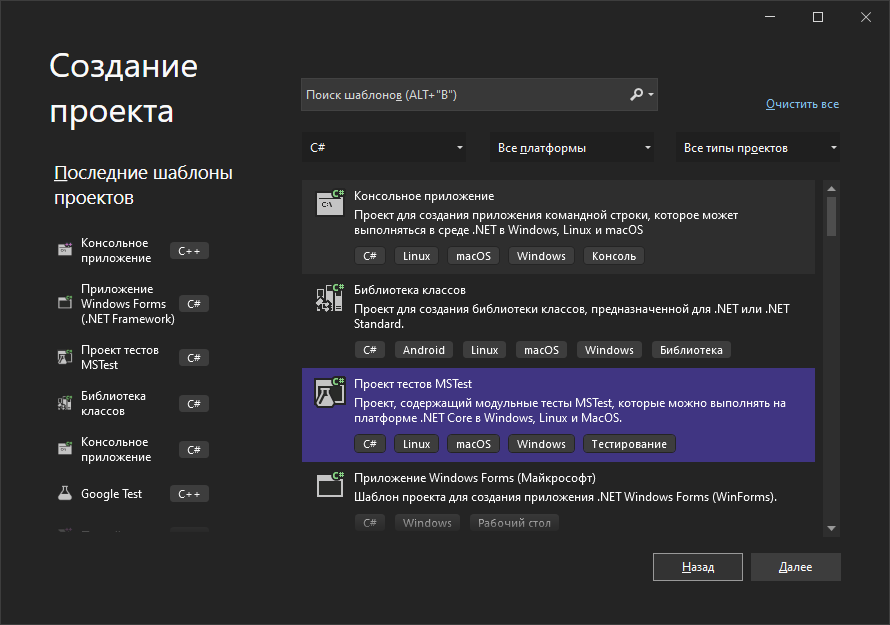
|  |  |
| --- | --- |
| Библиотека классов (универсальные приложения для Windows) | Создает управляемую библиотеку классов (DLL) на языке C#, которую могут использовать другие приложения UWP, написанные с применением управляемого кода. |
| Компонент среды выполнения Windows (универсальные приложения для Windows) | Создает [компонент среды выполнения Windows](https://learn.microsoft.com/ru-ru/windows/uwp/winrt-components/) на C#, который может использоваться в любом приложении UWP, независимо от того, на каком языке программирования такое приложение написано. |
| Необязательный пакет кода (универсальные приложения для Windows) | Создает дополнительный пакет с исполняемым кодом C#, который может загрузить приложение. Подробные сведения см. в статье [Дополнительные пакеты с исполняемым кодом](https://learn.microsoft.com/ru-ru/windows/msix/package/optional-packages-with-executable-code). |

Шаблон консольного приложения

Запустите Visual Studio.

На начальной странице выберите Создать проект.

На странице Создание проекта выберите C# из списка языков, а затем — Все платформы из списка платформ. Выберите шаблон Консольное приложение и нажмите кнопку Далее.



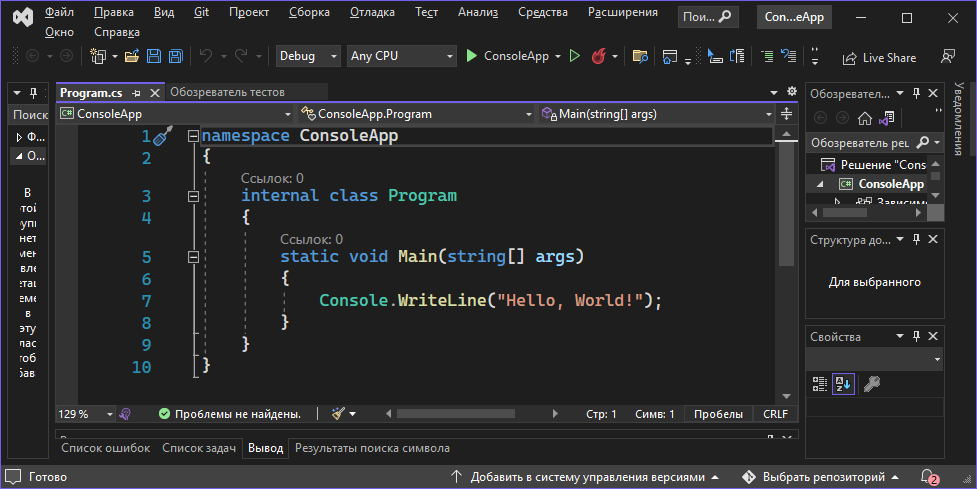
В диалоговом окне Настройка нового проекта введите название проекта в поле Имя проекта . Теперь щелкните Далее.

В диалоговом окне Дополнительные сведения выполните следующие действия.

Выберите .NET нужной версии.

Выберите «Не использовать [операторы верхнего уровня](https://learn.microsoft.com/ru-ru/dotnet/csharp/whats-new/tutorials/top-level-statements)».

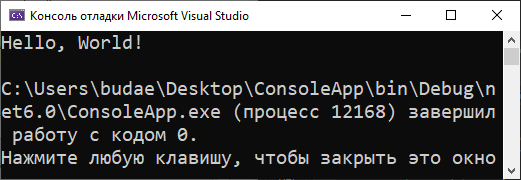
Нажмите кнопку создания.

Шаблон создает простое приложение, которое отображает сообщение "Hello, World!" в окне консоли. Код находится в файле Program.cs:

Код определяет класс Program с одним методом Main, который принимает массив String в качестве аргумента. Main — точка входа в приложение. Это метод, который автоматически вызывается средой выполнения при запуске приложения. Все аргументы, предоставленные в командной строке при запуске приложения, доступны через массив args.

Нажмите клавиши CTRL+F5, чтобы запустить программу без отладки.

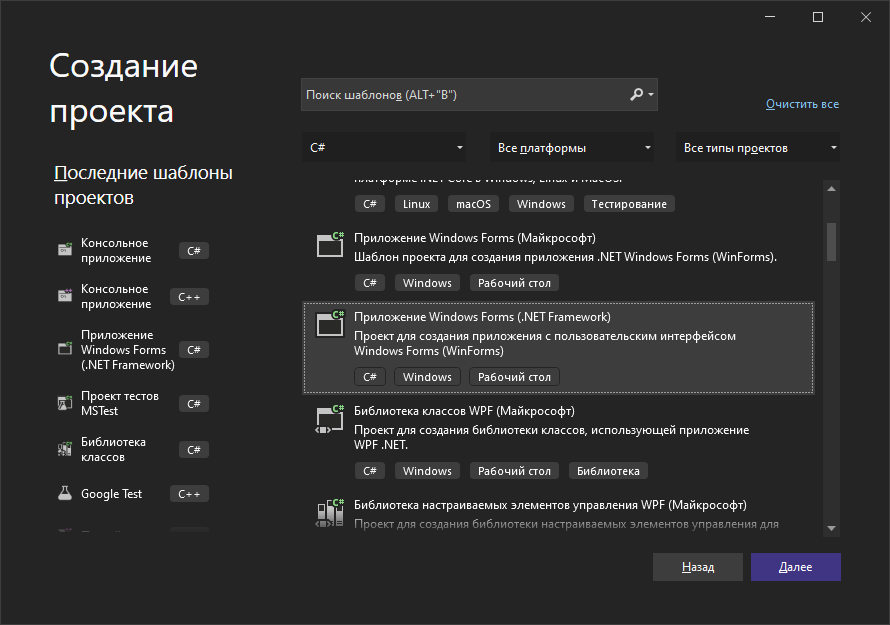
Откроется окно консоли с текстом "Hello, World!" на экране.



Для закрытия консольного окна нажмите любую клавишу.

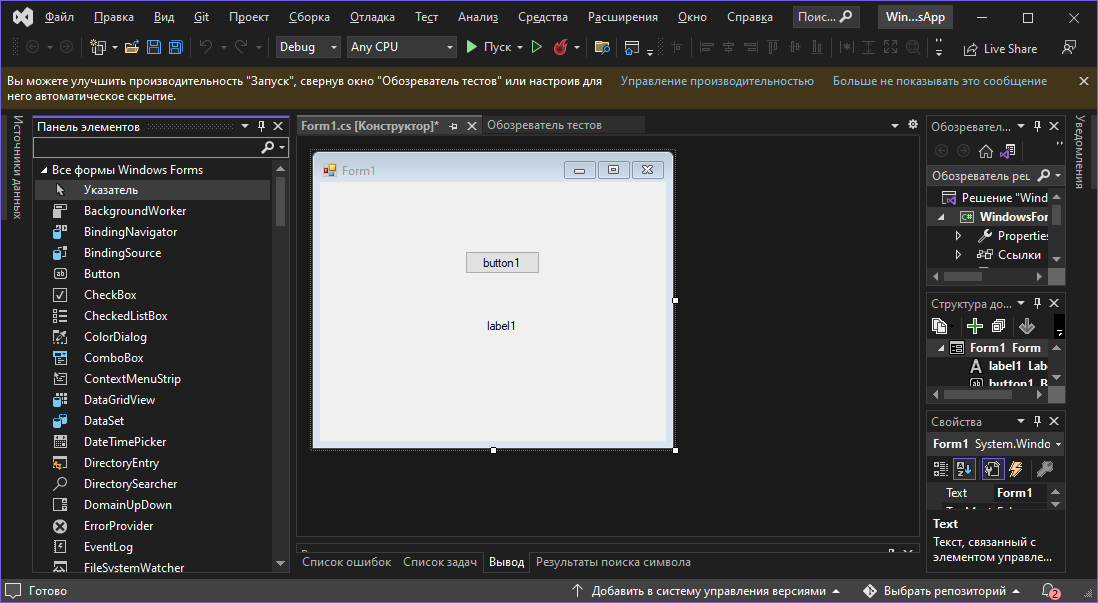
Шаблон приложения с GUI

На странице Создание проекта выберите C# из списка языков, а затем — Все платформы из списка платформ. Выберите шаблон Приложение Windows Forms (.NET Framework) и нажмите кнопку Далее.

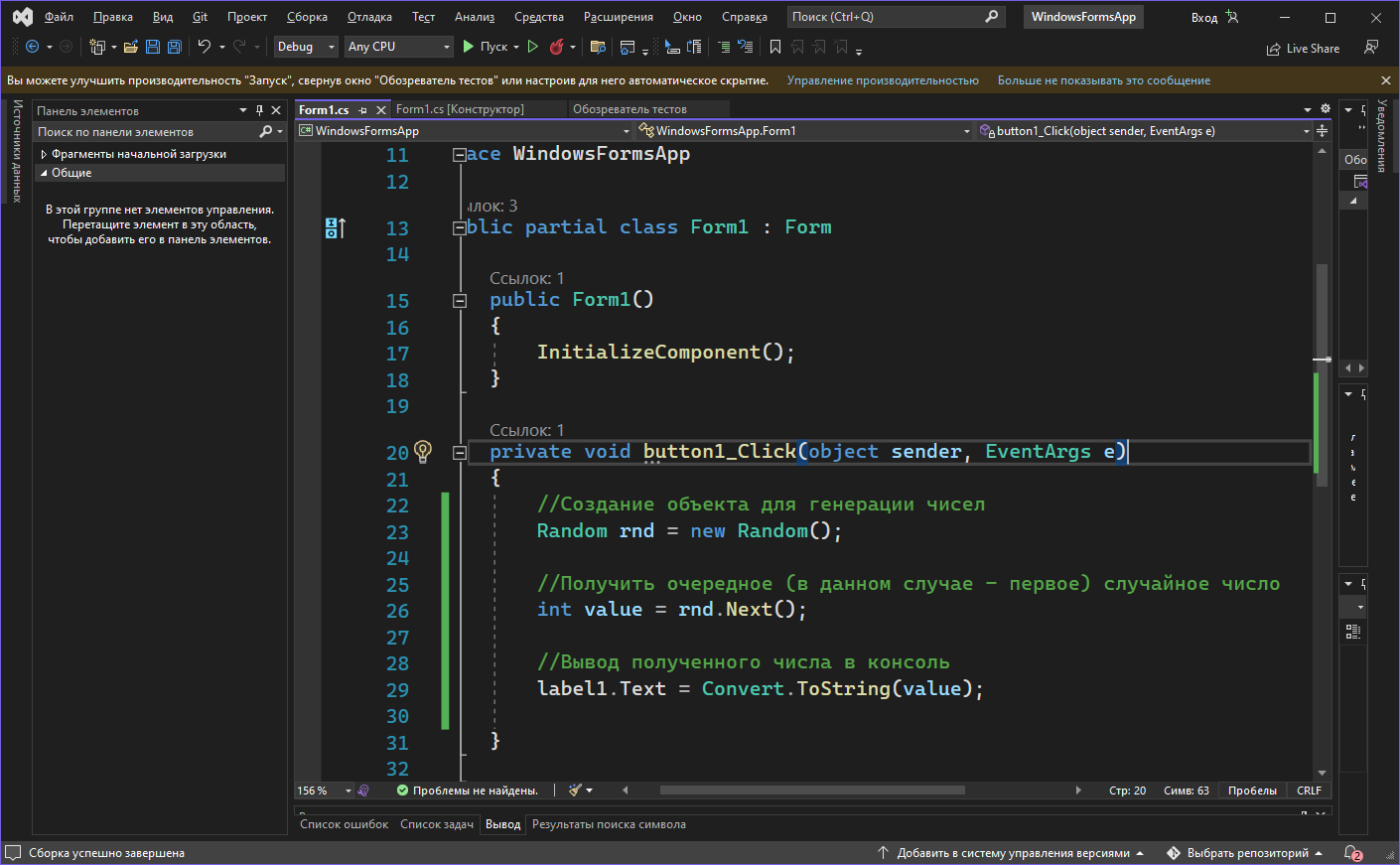


В следующем окне введите имя проекта и выберите необходимую версию .NET Framework

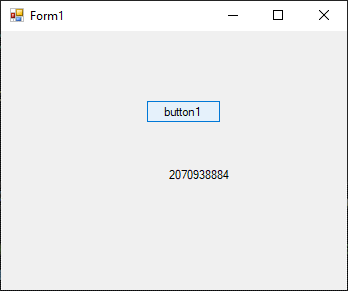
В появившемся окне перетащите из Панели элементов Button и Label.



Нажмите 2 раза на Кнопку. Откроется файл с обработчиками событий, в котором мы создадим вывод случайного числа по нажатии на кнопку.



Нажмите клавиши CTRL+F5, чтобы запустить программу без отладки. При нажатии на кнопку выводится случайное число



Среды разработки С#

***Visual Studio***

Описание: самая «правильная» среда разработки. С Visual Studio многие начинают знакомиться с языком и не расстаются с ней на протяжении всей карьеры программиста.

Плюсы:

* Официальная. Так как и язык, и среда разработки созданы в Microsoft, логично предположить, что ничего более функционального вы не найдете во всем Интернете. В некоторых случаях без Visual Studio не обойтись — например, при использовании технологий UWP и WPF.
* Бесплатная. Версии «Community edition» для рядового пользователя будет достаточно. Тем более, теперь можно подключать плагины (в отличие от старой версии Express).
* Функциональная. В Visual Studio множество качественных плагинов. С их помощью можно расширить функциональность приложения и подключить другие языки.
* Поддерживает платформы .NET. Visual Studio имеет широкие возможности по разработке приложений под Windows, в том числе в .NET-сегменте.
* Облачные хранилища. Зарегистрируйтесь в сообществе Visual Studio — и получите доступ к облачному хранилищу, где сможете располагать файлы проектов.
* Корпоративность. Технология бэклога позволяет членам команды взаимодействовать при гибкой методологии разработки.

Минусы:

* Баги при переходах с триал-версии. При переходе на платную версию могут теряться настройки и нарушаться работа корпоративного сервера.
* Сложность. Самостоятельно освоить Visual Studio новичку будет непросто — слишком много доступных функций, спрятанных в подразделах меню.

***Project Rider***

Описание: среда от JetBrains для работы с платформой .NET. Выпущена в прошлом году, но уже приобрела много поклонников.

Плюсы:

* ReSharper. Это плагин, изначально разработанный для повышения производительности Visual Studio. Теперь на его основе выпущена IDE.
* Поддержка полного цикла. Фирменная черта продуктов JetBrains, воплощенная и в Project Rider. С ним вы сможете организовать весь цикл создания ПО: от идеи до поддержки.
* Функциональность. Project Rider позволяет подключить MSBuild и XBuild, работать с CLI-проектами и организовать отладку приложений .NET and Mono. Множество опций для быстрого создания кода улучшает производительность.
* Multiple runtime. Поддержка нескольких запущенных программ.
* Кроссплатформенность. Project Rider работает с Windows, Linux и MacOS.
* Контроль версий. Встроенный инструмент позволяет напрямую организовать работу с Git, Mercurial и TFS.

Минусы:

* Молодость. Часть функциональности еще в разработке, не все стартовые баги исправлены.
* Стоимость. Самая дешевая версия Project Rider обойдется в 139 долларов за первый год использования. Но есть триал-версия и специальные предложения для студентов и непрофильных организаций.

***Eclipse***

Описание: одна из самых популярных мультиязычных сред. Ориентирована преимущественно на разработку Java-приложений, но полезна и для кодов на C#.

Плюсы:

* Множество плагинов. У Eclipse едва ли не самое большое число надстроек — «на все случаи жизни».
* Активное сообщество. Помогает быстрее освоить среду разработки, выпускает новые плагины.
* Отличные компилятор и отладчик. Первый работает на порядок быстрее, чем у конкурентов, второй — показывает потоки, пересечения, позволяет гибко управлять ходом отладки.
* Кастомизация. Благодаря плагинам и настройкам можно полностью персонализировать Eclipse.
* Бесплатность. Это open-source проект, абсолютно бесплатный.
* Высокая функциональность. Благодаря разработчикам-официалам и членам сообщества с помощью Eclipse можно провести любой C#-продукт по полному циклу разработки.

Минусы:

* Сложность. Как и любой функциональный продукт, Eclipse может показаться новичку слишком сложным.
* Нет гарантий надежности. Так как плагины создаются сообществом, за их качество отвечает только разработчик. Кроме того, сами создатели Eclipse с каждой новой версией плодят баги, не успевая порой исправлять старые.

Основные типы, особенности типизации, контейнеры

***Типы данных***

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.

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

bool alive = true;

bool isDead = false;

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

byte bit1 = 1;

byte bit2 = 102;

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

sbyte bit1 = -101;

sbyte bit2 = 102;

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

short n1 = 1;

short n2 = 102;

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

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:

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

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

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

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
* 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:

char a = 'A';

char b = '\x5A';

char c = '\u0420';

* string: хранит набор символов Unicode. Представлен системным типом System.String. Этому типу соответствуют строковые литералы.

string hello = "Hello";

string word = "world";

* object: может хранить значение любого типа данных и занимает 4 байта на 32-разрядной платформе и 8 байт на 64-разрядной платформе. Представлен системным типом System.Object, который является базовым для всех других типов и классов .NET.

object a = 22;

object b = 3.14;

object c = "hello code";

***Использование суффиксов***

При присвоении значений надо иметь в виду следующую тонкость: все вещественные литералы (дробные числа) рассматриваются как значения типа double. И чтобы указать, что дробное число представляет тип float или тип decimal, необходимо к литералу добавлять суффикс: F/f - для float и M/m - для decimal.

float a = 3.14F;

float b = 30.6f;

decimal c = 1005.8M;

decimal d = 334.8m;

Подобным образом все целочисленные литералы рассматриваются как значения типа int. Чтобы явным образом указать, что целочисленный литерал представляет значение типа uint, надо использовать суффикс U/u, для типа long - суффикс L/l, а для типа ulong - суффикс UL/ul:

uint a = 10U;

long b = 20L;

ulong c = 30UL;

***Использование системных типов***

Выше при перечислении всех базовых типов данных для каждого упоминался системный тип. Потому что название встроенного типа по сути представляет собой сокращенное обозначение системного типа. Например, следующие переменные будут эквивалентны по типу:

int a = 4;

System.Int32 b = 4;

***Неявная типизация***

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:

var hello = "Hell to World";

var c = 20;

Для неявной типизации вместо названия типа данных используется ключевое слово var. Затем уже при компиляции компилятор сам выводит тип данных исходя из присвоенного значения. Так как по умолчанию все целочисленные значения рассматриваются как значения типа int, то поэтому в итоге переменная c будет иметь тип int. Аналогично переменной hello присваивается строка, поэтому эта переменная будет иметь тип string

Эти переменные подобны обычным, однако они имеют некоторые ограничения.

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

// этот код работает

int a;

a = 20;

// этот код не работает

var c;

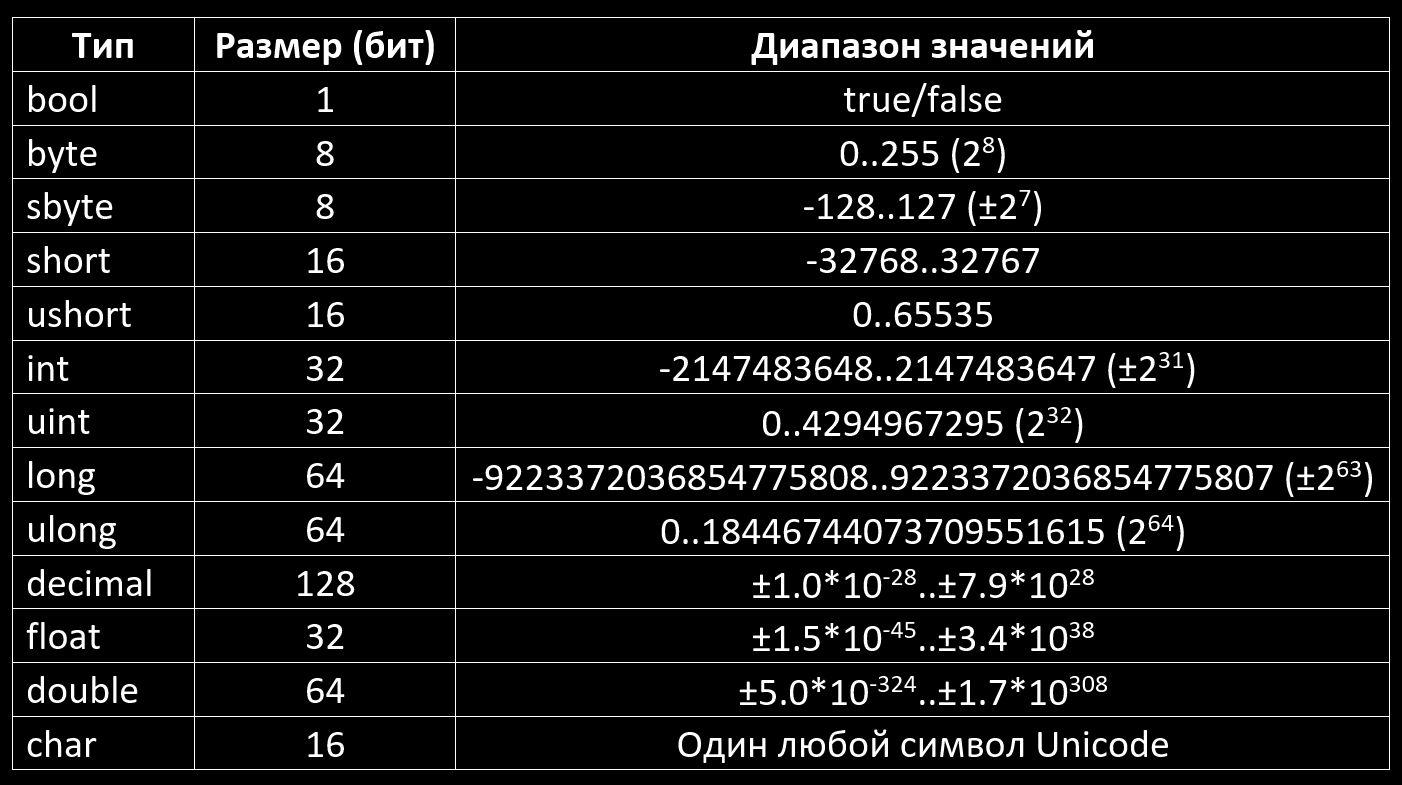
c= 20;

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

// этот код не работает

var c=null;

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



Общие сведения о классах

**Ссылочные типы**

Тип, определенный как, [class](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/class) является *ссылочным типом*. Во время выполнения при объявлении переменной ссылочного типа переменная содержит значение [null](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/null) , пока вы явным образом не создадите экземпляр класса с помощью [new](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/operators/new-operator) оператора или не назначьте ему объект совместимого типа, который мог быть создан в другом месте, как показано в следующем примере:

//Declaring an object of type MyClass.

MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.

MyClass mc2 = mc;

При создании объекта выделяется достаточный объем памяти для этого объекта в управляемой куче, и переменная хранит только ссылку на расположение данного объекта. Хранение типов в управляемой куче требует дополнительных действий как при выделении памяти, так и при удалении, которое выполняется функцией автоматического управления памятью в среде CLR, известной как *сборка мусора*. Сборка мусора является хорошо оптимизированным процессом и в большинстве случаев не создает помех для производительности. Дополнительные сведения о сборке мусора см. в разделе [Автоматическое управление памятью и сборка мусора](https://learn.microsoft.com/ru-ru/dotnet/standard/garbage-collection/fundamentals).

**Объявление классов**

Классы объявляются с помощью ключевого class слова , за которым следует уникальный идентификатор, как показано в следующем примере:

//[access modifier] - [class] - [identifier]

public class Customer

{

// Fields, properties, methods and events go here...

}

Ключевому слову class предшествует уровень доступа. Поскольку [public](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/public) используется в этом случае, любой пользователь может создать экземпляры этого класса. За именем класса следует ключевое слово class. Имя класса должно быть допустимым [именем идентификатора](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/coding-style/identifier-names) C#. Оставшаяся часть определения — это тело класса, в котором задаются данные и поведение. Поля, свойства, методы и события в классе собирательно называются *членами класса*.

**Создание объектов**

Несмотря на то, что они иногда взаимозаменяемы, класс и объект — разные вещи. Класс определяет тип объекта, но не является объектом. Объект — это конкретная сущность, основанная на классе, которую иногда называют экземпляром класса.

Объекты можно создать с помощью ключевого new слова , за которым следует имя класса, на основе которого будет основан объект, следующим образом:

Customer object1 = new Customer();

При создании экземпляра класса ссылка на объект передается программисту. В предыдущем примере object1 представляет собой ссылку на объект, который основан на Customer. Эта ссылка указывает на новый объект, но не содержит данные этого объекта. Фактически, можно создать ссылку на объект без создания собственно объекта:

Customer object2;

Мы не рекомендуем создавать ссылки на объекты, такие как предыдущие, которые не ссылаются на объект, так как попытка получить доступ к объекту через такую ссылку завершится ошибкой во время выполнения. Однако такую ссылку можно сделать указывающей на объект, создав новый объект или назначив ее существующему объекту, как показано далее:

Customer object3 = new Customer();

Customer object4 = object3;

В этом коде создаются две ссылки на объект, которые указывают на один и тот же объект. Таким образом, любые изменения объекта, выполненные посредством object3, отражаются при последующем использовании object4. Поскольку на объекты, основанные на классах, указывают ссылки, классы называют ссылочными типами.

**Наследование классов**

Классы полностью поддерживают *наследование*, фундаментальный механизм объектно ориентированного программирования. При создании класса можно наследовать от любого другого класса, который не определен как [sealed](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/sealed), а другие классы могут наследовать от класса и переопределять виртуальные методы класса. Кроме того, можно реализовать один или несколько интерфейсов.

При наследовании создается *производный* класс, то есть класс объявляется с помощью *базового класса*, от которого он наследует данные и поведение. Базовый класс задается добавлением после имени производного класса двоеточия и имени базового класса, как показано далее:

public class Manager : Employee

{

// Employee fields, properties, methods and events are inherited

// New Manager fields, properties, methods and events go here...

}

Когда класс объявляет базовый класс, он наследует все члены базового класса, за исключением конструкторов. Дополнительные сведения см. в разделе [Наследование](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/object-oriented/inheritance).

Класс в C# может напрямую наследовать только от одного базового класса. Тем не менее, поскольку базовый класс может сам наследовать от другого класса, класс может косвенно наследовать от нескольких базовых классов. Кроме того, класс может напрямую реализовать несколько интерфейсов. Дополнительные сведения см. в разделе [Интерфейсы](https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/types/interfaces).

Класс можно объявить [abstract](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/abstract). Абстрактный класс содержит абстрактные методы, которые имеют определение сигнатуры, но не имеют реализации. Нельзя создавать экземпляры абстрактных классов. Они могут использоваться только через производные классы, реализующие абстрактные методы. И наоборот, [запечатанный](https://learn.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/sealed) класс не позволяет другим классам быть от него производными. Дополнительные сведения см. в статье [Абстрактные и запечатанные классы и члены классов](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members).

Определения классов можно разделить между различными исходными файлами. Дополнительные сведения см. в разделе [Разделяемые классы и методы](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods).

**Пример**

В следующем примере определяется открытый класс, содержащий [автоматически реализуемое свойство](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/auto-implemented-properties), метод и специальный метод, который называется конструктором. Дополнительные сведения см. в статьях [о свойствах](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/properties), [методах](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/methods) и [конструкторах](https://learn.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/constructors) . Затем создаются экземпляры этого класса с помощью ключевого слова new.

using System;

public class Person

{

// Constructor that takes no arguments:

public Person()

{

Name = "unknown";

}

// Constructor that takes one argument:

public Person(string name)

{

Name = name;

}

// Auto-implemented readonly property:

public string Name { get; }

// Method that overrides the base class (System.Object) implementation.

public override string ToString()

{

return Name;

}

}

class TestPerson

{

static void Main()

{

// Call the constructor that has no parameters.

var person1 = new Person();

Console.WriteLine(person1.Name);

// Call the constructor that has one parameter.

var person2 = new Person("Sarah Jones");

Console.WriteLine(person2.Name);

// Get the string representation of the person2 instance.

Console.WriteLine(person2);

}

}

// Output:

// unknown

// Sarah Jones

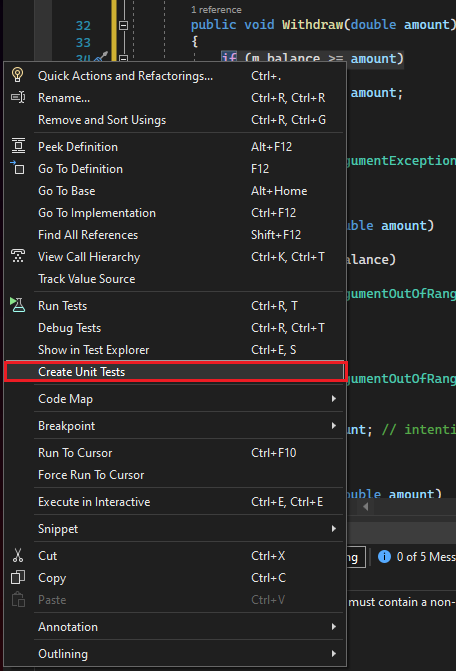
// Sarah Jones

Создание проекта модульного теста

Для C# как правило, проще создать проект модульного теста и заглушки модульных тестов из кода. Кроме того, можно создать проект модульных тестов и тесты вручную в зависимости от потребностей. Если вы хотите создавать модульные тесты из кода на сторонней платформе, вам потребуется установить одно из этих расширений: NUnit или xUnit. Если вы не используете C#, пропустите этот раздел и перейдите к разделу Создание проекта и модульных тестов вручную.

**Создание проекта модульного теста и заглушек модульных тестов**

1. В окне редактора кода выберите в контекстном меню команду Создать модульные тесты.

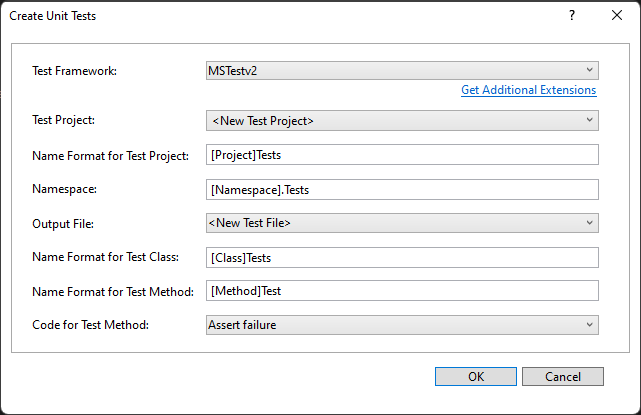


В окне редактора просмотрите контекстное меню

*Примечание*

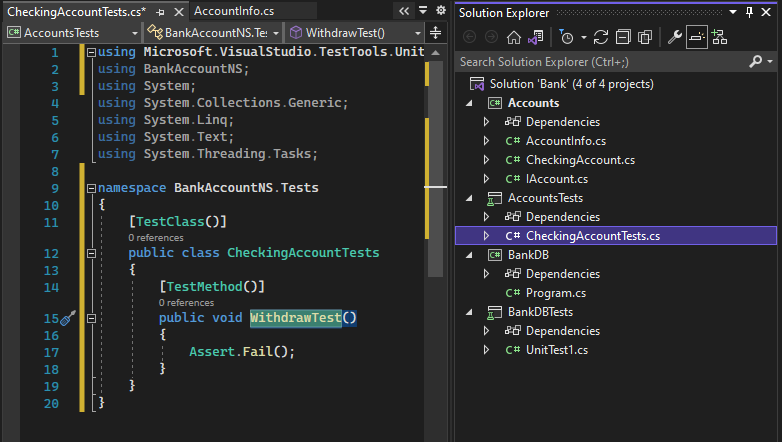
*Команда меню Создать модульные тесты доступна только для кода C#. Чтобы использовать этот метод для .NET Core или .NET Standard, требуется Visual Studio 2019 или более поздней версии.*

1. Нажмите кнопку ОК, чтобы принять значения по умолчанию для создания модульных тестов, или измените значения, которые использовались для создания и назначения имени проекта модульного теста и модульных тестов. Можно выбрать код, который добавляется по умолчанию в методы модульных тестов.



Диалоговое окно

1. Заглушки модульных тестов создаются в новом проекте модульного теста для всех методов в классе.



Модульные тесты созданы

Литература

[**https://learn.microsoft.com/ru-ru/windows/apps/desktop/visual-studio-templates**](https://learn.microsoft.com/ru-ru/windows/apps/desktop/visual-studio-templates)

[**https://learn.microsoft.com/ru-ru/dotnet/csharp/whats-new/csharp-version-history**](https://learn.microsoft.com/ru-ru/dotnet/csharp/whats-new/csharp-version-history)

[**https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/**](https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/)

[**http://bourabai.ru/alg/c-sharp.htm#:~:text=C%23%20%D0%BE%D1%82%D0%BD%D0%BE%D1%81%D0%B8%D1%82%D1%81%D1%8F%20%D0%BA%20%D1%81%D0%B5%D0%BC%D1%8C%D0%B5%20%D1%8F%D0%B7%D1%8B%D0%BA%D0%BE%D0%B2,%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D1%8F%2C%20%D0%BA%D0%BE%D0%BC%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%80%D0%B8%D0%B8%20%D0%B2%20%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B5%20XML**](http://bourabai.ru/alg/c-sharp.htm#:~:text=C%23%20%D0%BE%D1%82%D0%BD%D0%BE%D1%81%D0%B8%D1%82%D1%81%D1%8F%20%D0%BA%20%D1%81%D0%B5%D0%BC%D1%8C%D0%B5%20%D1%8F%D0%B7%D1%8B%D0%BA%D0%BE%D0%B2,%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D1%8F%2C%20%D0%BA%D0%BE%D0%BC%D0%BC%D0%B5%D0%BD%D)**.**

[**https://learn.microsoft.com/ru-ru/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2022**](https://learn.microsoft.com/ru-ru/visualstudio/test/walkthrough-creating-and-running-unit-tests-for-managed-code?view=vs-2022)

[**https://habr.com/ru/post/350708/**](https://habr.com/ru/post/350708/)

[**https://metanit.com/sharp/windowsforms/3.1.php**](https://metanit.com/sharp/windowsforms/3.1.php)

[**https://habr.com/ru/post/161205/**](https://habr.com/ru/post/161205/)

[**https://tyapk.ru/blog/post/programming-languages-typing**](https://tyapk.ru/blog/post/programming-languages-typing)

[**https://metanit.com/sharp/tutorial/2.1.php**](https://metanit.com/sharp/tutorial/2.1.php)

**https://gb.ru/posts/c\_sharp\_ides**