

Министерство образования Республики Беларусь
«ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. ЕВФРОСИНИИ
ПОЛОЦКОЙ»

Факультет информационных технологий
Кафедра технологий программирования

**Методические указания для выполнения
лабораторной работы №8
по курсу «Конструирование программного
обеспечения»**

«Работа со строковыми данными в языке высокого уровня»

Полоцк, 2022 г.

ЦЕЛЬ РАБОТЫ

Познакомится с таким понятием как строка. Разобрать методы для работы со строками в C++ и C#. На основе примеров, приведенных в данной лабораторной работе, выполнить свой вариант практического задания.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Строковые данные в C#

Довольно большое количество задач, которые могут встретиться при разработке приложений, так или иначе связано с обработкой строк - парсинг веб-страниц, поиск в тексте, какие-то аналитические задачи, связанные с извлечением нужной информации из текста и т.д. Поэтому в этом плане работе со строками уделяется особое внимание.

В языке C# строковые значения представляет тип `string`, а вся функциональность работы с данным типом сосредоточена в классе `System.String`. Собственно, `string` является псевдонимом для класса `String`. Объекты этого класса представляют текст как последовательность символов Unicode. Максимальный размер объекта `String` может составлять в памяти 2 ГБ, или около 1 миллиарда символов.

Создание строк

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

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

Console.WriteLine(s1); // hello
Console.WriteLine(s2); // aaaaaa
Console.WriteLine(s3); // world
Console.WriteLine(s4); // orl
```

Конструктор `String` имеет различное число версий. Так, вызов конструктора

```
new String('a', 6)
```

6 раз повторит объект из первого параметра, то есть фактически создаст строку "aaaaaa".

Еще один конструктор принимает массив символов, из которых создается строка

```
string s3 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' });
```

Третий использованный выше в примере конструктор позволяет создать строку из части массива символов. Вторым параметром передается начальный индекс, с которого извлекаются символы, а третий параметр указывает на количество символов:

```
string s4 = new String(new char[] { 'w', 'o', 'r', 'l', 'd' }, 1, 3); // orl
```

Строка как набор символов

Так как строка хранит коллекцию символов, в ней определен индексатор для доступа к этим символам:

```
public char this[int index] {get;}
```

Применяя индексатор, мы можем обратиться к строке как к массиву символов и получить по индексу любой из ее символов:

```
string message = "hello";  
// получаем символ  
char firstChar = message[1]; // символ 'e'  
Console.WriteLine(firstChar); //e  
  
Console.WriteLine(message.Length); // длина строки
```

Используя свойство `Length`, как и в обычном массиве, можно получить длину строки.

Перебор строк

Класс `String` реализует интерфейс `IEnumerable`, благодаря чему строку можно перебрать в цикле `foreach` как набор объектов `char`. Также можно с помощью других типов циклов перебрать строку, применяя обращение к символам по индексу:

```
string message = "hello";  
for(var i = 0; i < message.Length; i++)  
{
```

```
        Console.WriteLine(message[i]);  
    }  
    foreach(var ch in message)  
    {  
        Console.WriteLine(ch);  
    }  
}
```

Сравнение строк

В отличие от других классов строки сравниваются по значению их символов, а не по ссылкам:

```
string message1 = "hello";  
string message2 = "hello";  
Console.WriteLine(message1 == message2);    // true
```

Основные методы строк

Основная функциональность класса `String` раскрывается через его методы, среди которых можно выделить следующие:

- **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 строки
- **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

Объединение строк

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

```
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:

```
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);
Console.WriteLine(s10); // apple a day keeps a doctor away
```

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

Сравнение строк

Для сравнения строк применяется статический метод Compare:

```
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.

В данном случае так как символ `h` по алфавиту стоит выше символа `w`, то и первая строка будет стоять выше.

Поиск в строке

С помощью метода `IndexOf` мы можем определить индекс первого вхождения отдельного символа или подстроки в строке:

```

string s1 = "hello world";
char ch = 'o';
int indexOfChar = s1.IndexOf(ch); // равно 4
Console.WriteLine(indexOfChar);

string substring = "wor";
int indexOfSubstring = s1.IndexOf(substring); // равно 6
Console.WriteLine(indexOfSubstring);

```

Подобным образом действует метод `LastIndexOf`, только находит индекс последнего вхождения символа или подстроки в строку.

Еще одна группа методов позволяет узнать начинается ли строка на определенную подстроку. Для этого предназначены методы `StartsWith` и `EndsWith`. Например, в массиве строк хранится список файлов, и нам надо вывести все файлы с расширением `exe`:

```

var files = new string[]
{
    "myapp.exe",
    "forest.jpg",
    "main.exe",
    "book.pdf",
    "river.png"
};

```

```
for (int i = 0; i < files.Length; i++)
{
    if (files[i].EndsWith(".exe"))
        Console.WriteLine(files[i]);
}
```

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. Например, подсчитаем количество слов в строке, разделив ее по пробельным символам:

```
string text = "И поэтому все так произошло";

string[] words = text.Split(new char[] { ' ' });

foreach (string s in words)
{
    Console.WriteLine(s);
}
```

Это не лучший способ разделения по пробелам, так как во входной строке у нас могло бы быть несколько подряд идущих пробелов и в итоговый массив также бы попадали пробелы, поэтому лучше использовать другую версию метода:

```
string[] words = text.Split(new char[] { ' ' },
StringSplitOptions.RemoveEmptyEntries);
```

Второй параметр `StringSplitOptions.RemoveEmptyEntries` говорит, что надо удалить все пустые подстроки.

Обрезка строки

Для обрезки начальных или конечных символов используется функция `Trim`:

```
string text = " hello world ";

text = text.Trim(); // результат "hello world"
text = text.Trim(new char[] { 'd', 'h' }); // результат "ello
worl"
```

Функция `Trim` без параметров обрезает начальные и конечные пробелы и возвращает обрезанную строку. Чтобы явным образом указать, какие начальные и конечные символы следует обрезать, мы можем передать в функцию массив этих символов.

Эта функция имеет частичные аналоги: функция `TrimStart` обрезает начальные символы, а функция `TrimEnd` обрезает конечные символы.

Обрезать определенную часть строки позволяет функция `Substring`:

```
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.

Вставка

Для вставки одной строки в другую применяется функция `Insert`:

```
string text = "Хороший день";  
string substring = "замечательный ";  
  
text = text.Insert(8, substring);  
Console.WriteLine(text);    // Хороший замечательный день
```

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`:

```
string text = "Хороший день";  
// индекс последнего символа  
int ind = text.Length - 1;  
// вырезаем последний символ
```



```
text = text.Remove(ind);  
Console.WriteLine(text);    // Хороший ден  
  
// вырезаем первые два символа  
text = text.Remove(0, 2);  
Console.WriteLine(text);    // роший ден
```

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.

Замена

Чтобы заменить один символ или подстроку на другую, применяется метод `Replace`:

```
string text = "хороший день";  
  
text = text.Replace("хороший", "плохой");  
Console.WriteLine(text);    // плохой день  
  
text = text.Replace("о", "");  
Console.WriteLine(text);    // плхй день
```

Во втором случае применения функции `Replace` строка из одного символа "о" заменяется на пустую строку, то есть фактически удаляется из текста. Подобным способом легко удалять какой-то определенный текст в строках.

Смена регистра

Для приведения строки к верхнему и нижнему регистру используются соответственно функции `ToUpper()` и `ToLower()`:

```
string hello = "Hello world!";  
Console.WriteLine(hello.ToLower()); // hello world!  
Console.WriteLine(hello.ToUpper()); // HELLO WORLD!
```

Форматирование строк

При выводе строк в консоли с помощью метода `Console.WriteLine` для встраивания значений в строку мы можем применять форматирование вместо конкатенации:

```
string name = "Tom";
```

```
int age = 23;
Console.WriteLine("Имя: {0}  Возраст: {1}", name, age);
// консольный вывод
// Имя: Tom  Возраст: 23
```

В строке "Имя: {0} Возраст: {1}" на место {0} и {1} затем будут вставляться в порядке следования значения переменных name и age.

То же самое форматирование в строке мы можем сделать не только в методе Console.WriteLine, но и в любом месте программы с помощью метода string.Format:

```
string name = "Tom";
int age = 23;
string output = string.Format("Имя: {0}  Возраст: {1}", name,
age);
Console.WriteLine(output);
```

Метод Format принимает строку с плейсхолдерами типа {0}, {1} и т.д., а также набор аргументов, которые вставляются на место данных плейсхолдеров. В итоге генерируется новая строка.

Спецификаторы форматирования

В методе Format могут использоваться различные спецификаторы и описатели, которые позволяют настроить вывод данных. Рассмотрим основные описатели. Все используемые форматы:

C / c	Задаёт формат денежной единицы, указывает количество десятичных разрядов после запятой
D / d	Целочисленный формат, указывает минимальное количество цифр
E / e	Экспоненциальное представление числа, указывает количество десятичных разрядов после запятой
F / f	Формат дробных чисел с фиксированной точкой, указывает количество десятичных разрядов после запятой
G / g	Задаёт более короткий из двух форматов: F или E
N / n	Также задаёт формат дробных чисел с фиксированной точкой, определяет количество разрядов после запятой
P / p	Задаёт отображения знака процентов рядом с числом, указывает количество десятичных разрядов после запятой
X / x	Шестнадцатеричный формат числа

Форматирование валюты

Для форматирования валюты используется описатель «C»:

```
double number = 23.7;
string result = string.Format("{0:C0}", number);
Console.WriteLine(result); // 24 p.
string result2 = string.Format("{0:C2}", number);
Console.WriteLine(result2); // 23,70 p.
```

Число после описателя указывает, сколько чисел будет использоваться после разделителя между целой и дробной частью. При выводе также добавляется обозначение денежного знака для текущей культуры компьютера. В зависимости от локализации текущей операционной системы результат может различаться. Также обратите внимание на округление в первом примере.

Форматирование целых чисел

Для форматирования целочисленных значение применяется описатель «d»:

```
int number = 23;
string result = string.Format("{0:d}", number);
Console.WriteLine(result); // 23
string result2 = string.Format("{0:d4}", number);
Console.WriteLine(result2); // 0023
```

Число после описателя указывает, сколько цифр будет в числовом значении. Если в исходном числе цифр меньше, то к нему добавляются нули.

Форматирование дробных чисел

Для форматирования дробны чисел используется описатель «F», число после которого указывает, сколько знаков будет использоваться после разделителя между целой и дробной частью. Если исходное число - целое, то к нему добавляются разделитель и нули.

```
int number = 23;
string result = string.Format("{0:f}", number);
Console.WriteLine(result); // 23,00

double number2 = 45.08;
string result2 = string.Format("{0:f4}", number2);
Console.WriteLine(result2); // 45,0800
```

```
double number3 = 25.07;  
string result3 = string.Format("{0:f1}", number3);  
Console.WriteLine(result3); // 25,1
```

Формат процентов

Описатель "P" задает отображение процентов. Используемый с ним числовой спецификатор указывает, сколько знаков будет после запятой:

```
decimal number = 0.15345m;  
Console.WriteLine("{0:P1}", number); // 15,3%
```

Настраиваемые форматы

Используя знак #, можно настроить формат вывода. Например, нам надо вывести некоторое число в формате телефона +x (xxx)xxx-xx-xx:

```
long number = 19876543210;  
string result = string.Format("{0:+# (###) ###-##-##}", number);  
Console.WriteLine(result); // +1 (987) 654-32-10
```

Метод ToString

Метод ToString() не только получает строковое описание объекта, но и может осуществлять форматирование. Он поддерживает те же описатели, что используются в методе Format:

```
long number = 19876543210;  
Console.WriteLine(number.ToString("+# (###) ###-##-##")); // +1  
(987) 654-32-10  
  
double money = 24.8;  
Console.WriteLine(money.ToString("C2")); // 24,80 p.
```

Интерполяция строк

Интерполяция строк призвана упростить форматирование строк. Так, перепишем пример с выводом значений переменных в строке:

```
string name = "Tom";  
int age = 23;  
  
Console.WriteLine($"Имя: {name}  Возраст: {age}");  
// консольный вывод  
// Имя: Tom  Возраст: 23
```

Знак доллара перед строкой указывает, что будет осуществляться интерполяция строк. Внутри строки опять же используются плейсхолдеры {...}, только внутри фигурных скобок уже можно напрямую писать те выражения, которые мы хотим вывести.

Интерполяция по сути представляет более лаконичное форматирование. При этом внутри фигурных скобок мы можем указывать не только свойства, но и различные выражения языка C#:

```
int x = 8;
int y = 7;
string result = $"{x} + {y} = {x + y}";
Console.WriteLine(result); // 8 + 7 = 15
```

Также внутри фигурных скобок можно выполнять более сложные выражения, например, вызывать методы:

```
int x = 8;
int y = 7;
string result = $"{x} * {y} = {Multiply(x, y)}";
Console.WriteLine(result); // 8 * 7 = 56
int Multiply(int a, int b) => a * b;
```

Уже внутри строки можно применять форматирование. В этом случае мы можем применять все те же описатели, что и в методе `Format`. Например, выведем номер телефона в формате +x xxx-xxx-xx-xx:

```
long number = 19876543210;
Console.WriteLine($"{number:+# ### ## ##}"); // +1 987 654 32
10
```

Добавляем пространство до и после форматируемого вывода:

```
string name = "Tom";
int age = 23;

Console.WriteLine($"Имя: {name, -5} Возраст: {age}"); // пробелы
после
Console.WriteLine($"Имя: {name, 5} Возраст: {age}"); // пробелы
до
```

```
Имя: Том    Возраст: 23
Имя:   Том Возраст: 23
```

Класс `StringBuilder`

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 первого символа - 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.

Создание `StringBuilder`

Для создания объекта `StringBuilder` можно использовать ряд его конструкторов. Прежде всего можно создать пустой `StringBuilder`:

```
using System.Text;
StringBuilder sb = new StringBuilder();
```

Можно сразу инициализировать объект определенной строкой:

```
StringBuilder sb = new StringBuilder("Привет мир");
```

С помощью метода `ToString()` мы можем получить строку, которая хранится в `StringBuilder`:

```
var sb = new StringBuilder("Hello World");
Console.WriteLine(sb.ToString());    // Hello World
```

Либо можно просто передать объект `StringBuilder`:

```
var sb = new StringBuilder("Hello World");
Console.WriteLine(sb);    // Hello World
```

Длина и емкость `StringBuilder`

Для хранения длины строки в классе `StringBuilder` определено свойство `Length`. Однако есть и вторая величина - емкость выделенной памяти. Это значение

хранится в свойстве `Capacity`. Емкость - это выделенная память под объект. Установка емкости позволяет уменьшить выделения памяти и тем самым повысить производительность.

Если строка, которая передается в конструктор `StringBuilder`, имеет длину 16 символов или меньше, то начальная ёмкость в `StringBuilder` равна 16. Если начальная строка больше 16 символов, то в качестве начальной емкости `StringBuilder` будет использовать длину строки.

Например, посмотрим, что содержат данные свойства:

```
using System.Text;

StringBuilder sb = new StringBuilder("Привет мир");
Console.WriteLine($"Длина: {sb.Length}");           // Длина: 10
Console.WriteLine($"Емкость: {sb.Capacity}");       // Емкость: 16
```

Хотя в данном случае длина равна 10 символов, но реально емкость будет составлять по умолчанию 16 символов. То есть мы видим, что при создании строки `StringBuilder` выделяет памяти больше, чем необходимо этой строке. При увеличении строки в `StringBuilder`, когда количество символов превосходит начальную емкость, то емкость увеличивается в два и более раз.

Если у нас заранее известен максимальный размер объекта, то мы можем таким образом сразу задать емкость с помощью одного из конструкторов и тем самым избежать последующих издержек при дополнительном выделении памяти.

```
var sb = new StringBuilder(32);
```

`StringBuilder` также позволяет сразу задать строку и емкость:

```
var sb = new StringBuilder("Привет мир", 32);
```

Операции со строками в `StringBuilder`

Для операций над строками класс `StringBuilder` определяет ряд методов:

- `Append`: добавляет подстроку в объект `StringBuilder`
- `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 подстроку в конец объекта `StringBuilder`

Теперь посмотрим на примере метода Append() использование и преимущества класса StringBuilder:

```
using System.Text;

var sb = new StringBuilder("Название: ");
Console.WriteLine(sb);    // Название:
Console.WriteLine($"Длина: {sb.Length}"); // 10
Console.WriteLine($"Емкость: {sb.Capacity}"); // 16

sb.Append(" Руководство");
Console.WriteLine(sb);    // Название: Руководство
Console.WriteLine($"Длина: {sb.Length}"); // 22
Console.WriteLine($"Емкость: {sb.Capacity}"); // 32

sb.Append(" по C#");
Console.WriteLine(sb);    // Название: Руководство по C#
Console.WriteLine($"Длина: {sb.Length}"); // 28
Console.WriteLine($"Емкость: {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.

Используем остальные методы StringBuilder:

```
var sb = new StringBuilder("Привет мир");
sb.Append("!");
sb.Insert(7, "компьютерный ");
Console.WriteLine(sb);    // Привет компьютерный мир!

// заменяем слово
sb.Replace("мир", "world");
Console.WriteLine(sb);    // Привет компьютерный world!

// удаляем 13 символов, начиная с 7-го
sb.Remove(7, 13);
Console.WriteLine(sb);    // Привет world!

// получаем строку из объекта StringBuilder
string text = sb.ToString();
Console.WriteLine(text);    // Привет world!
```


Когда надо использовать класс `String`, а когда `StringBuilder`?

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.

Регулярные выражения

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;

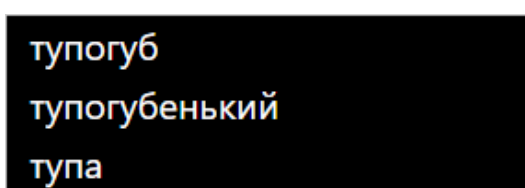
string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела  
была тупа";
Regex regex = new Regex(@"тип(\w*)");
MatchCollection matches = regex.Matches(s);
if (matches.Count > 0)
{
    foreach (Match match in matches)
        Console.WriteLine(match.Value);
}
else
{
    Console.WriteLine("Совпадений 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(@"тип(\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.

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»`.

Строковые данные в C++

Для хранения строк в C++ применяется тип `string`. Для использования этого типа его необходимо подключить в код с помощью директивы `include`:

```
#include <string>
#include <iostream>

int main()
{
    std::string hello = "Hello World!";
    std::cout << hello << "\n";
    return 0;
}
```

Тип `string` определен в стандартной библиотеке и при его использовании надо указывать пространство имен `std`.

Либо можно использовать выражение `using`, чтобы не указывать префикс `std`:

```
using std::string;
```

В данном случае значение переменной `hello`, которая представляет тип `string`, выводится на консоль.

При компиляции через `g++` может потребоваться указать флаг `-static`. То есть если код определен в файл `hello.cpp`, то команда на компиляцию для `g++` может выглядеть следующим образом:

```
g++ hello.cpp -o hello -static
```

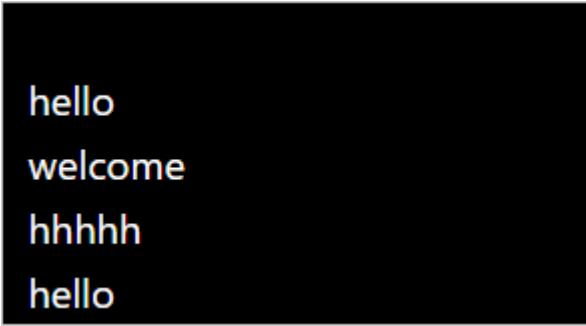
Для инициализации строк можно использовать различные способы:

```
#include <string>
#include <iostream>

int main()
{
    std::string s1;                // пустая строка
    std::string s2 = "hello";      // hello
    std::string s3("welcome");     // welcome
    std::string s4(5, 'h');        // hhhh
    std::string s5 = s2;           // hello

    std::cout << s1 << "\n";
    std::cout << s2 << "\n";
    std::cout << s3 << "\n";
    std::cout << s4 << "\n";
    std::cout << s5 << "\n";
    return 0;
}
```

Консольный вывод данной программы:



```
hello
welcome
hhhh
hello
```

Если при определении переменной типа `string` мы не присваиваем ей никакого значения, то по умолчанию данная переменная содержит пустую строку:

```
std::string s1;
```

Также можно инициализировать переменную строчным литералом, который заключается в двойные кавычки:

```
std::string s2 = "hello";
```

В качестве альтернативы можно передавать строку в скобках после определения переменной:

```
std::string s3("welcome");
```

Если необходимо, чтобы строка содержала определенное количество определенных символов, то можно указать в скобках количество символов и сам символ:

```
std::string s4(5, 'h');
```

И также можно передать переменной копию другой строки:

```
std::string s5 = s2;
```

Конкатенация строк

Над строками можно выполнять ряд операций. В частности, можно объединять строки с помощью стандартной операции сложения:

```
#include <iostream>
using std::cout;
using std::endl;
using std::string;

int main()
{
    string s1 = "hello";
    string s2 = "world";
    string s3 = s1 + " " + s2; // hello world
    cout << s3 << endl;
    return 0;
}
```

Сравнение строк

К строкам можно применять операции сравнения. Оператор `==` возвращает `true`, если все символы обеих строк равны.

```
std::string s1 = "hello";
std::string s2 = "world";

bool result = s1 == s2;           // false
result = s1 == "Hello";          // false
```



```
result = s1 == "hello";      // true
```

При этом символы должны совпадать в том числе по регистру.

Операция `!=` возвращает `true`, если две строки не совпадают.

```
std::string s1 = "hello";  
std::string s2 = "world";
```

```
bool result = s1 != s2;      // true  
result = s1 != "Hello";     // true  
result = s1 != "hello";     // false
```

Остальные базовые операции сравнения `<`, `<=`, `>`, `>=` сравнивают строки в зависимости от регистра и алфавитного порядка символов. Например, строка "b" условно больше строки "a", так как символ b по алфавиту идет после символа a. А строка "a" больше строки "A". Если первые символы строки равны, то сравниваются последующие символы:

```
std::string s1 = "Aport";  
std::string s2 = "Apricot";  
bool result = s1 > s2;      // false
```

В данном случае условие `s1 > s2` ложно, то есть `s2` больше чем `s1`, так как при равенстве первых двух символов ("Ap") третий символ второй строки ("o") стоит в алфавите до третьего символа второй строки ("r"), то есть "o" меньше чем "r".

Размер строки

С помощью метода `size()` можно узнать размер строки, то есть из скольких символов она состоит:

```
std::string s1 = "hello";  
std::cout << s1.size() << std::endl;    // 5
```

Если строка пустая, то она содержит 0 символов. В этом случае мы можем применить метод `empty()` - он возвращает `true`, если строка пустая:

```
std::string s1 = "";  
if(s1.empty())  
    std::cout << "String is empty" << std::endl;
```

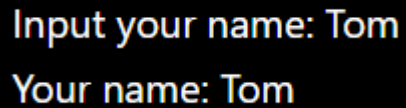
Чтение строки с консоли

Для считывания введенной строки с консоли можно использовать объект `std::cin`:

```
#include <iostream>
#include <string>

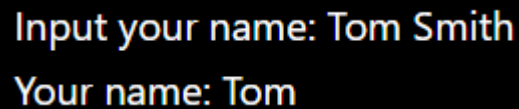
int main()
{
    std::string name;
    std::cout << "Input your name: ";
    std::cin >> name;
    std::cout << "Your name: " << name << std::endl;
    return 0;
}
```

Консольный вывод:



```
Input your name: Tom
Your name: Tom
```

Однако если при данном способе ввода строка будет содержать подстроки, разделенные пробелом, то `std::cin` будет использовать только первую подстроку:



```
Input your name: Tom Smith
Your name: Tom
```

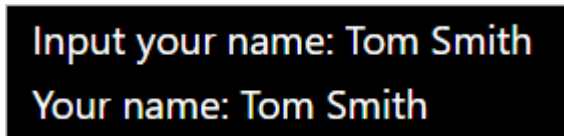
Чтобы считать всю строку, применяется метод `getline()`:

```
#include <iostream>
#include <string>

int main()
{
    std::string name;
    std::cout << "Input your name: ";
    getline(std::cin, name);
    std::cout << "Your name: " << name << std::endl;
    return 0;
}
```

Метод `getline` принимает два объекта - `std::cin` и переменную, в которую надо считать строку.

Консольный вывод:



```
Input your name: Tom Smith
Your name: Tom Smith
```

Получение и изменение символов строки

Подобно массиву мы можем обращаться с помощью индексов к отдельным символам строки, получать и изменять их:

```
std::string hello = "Hello";
char c = hello[1];      // e
hello[0]='M';
std::cout << hello << std::endl;    // Mello
```

Символьные массивы

Массив символов, последний элемент которого представляет нулевой символ `'\0'`, может использоваться как строка:

```
#include <iostream>

int main()
{
    char letters[] = {'h', 'e', 'l', 'l', 'o', '\0'};
    std::cout << letters << std::endl;

    return 0;
}
```

Данный код выведет на консоль строку «hello». Подобное определение массива строк будет также эквивалентно следующему:

```
char letters[] = "hello";
```

Однако подобное использование массива строк унаследовано от языка Си, а при написании программ на C++ при работе со строками следует отдавать предпочтение встроенному типу `string`, а не массиву символов.

Содержание отчета

Отчет должен включать:

- а) титульный лист;
- б) формулировку цели работы;
- в) описание результатов выполнения заданий:
 - листинги программ;
 - результаты выполнения программ;
- г) выводы, согласованные с целью работы.

Варианты

Вариант	Задание 1	Задание 2	Задание 3	Задание 4
1	5	16	27	32
2	6	14	23	39
3	7	12	26	35
4	8	13	21	34
5	9	17	22	31
6	10	15	28	33
7	1	11	25	37
8	2	19	30	38
9	3	18	24	40
10	4	20	29	36

Задания

1. Дана строка. Вывести ее три раза через запятую и показать количество символов в ней.
2. Дана строка. Вывести первый, последний и средний (если он есть)) символы.
3. Дана строка. Вывести первые три символа и последний три символа, если длина строки больше 5. Иначе вывести первый символ столько раз, какова длина строки.
4. Сформировать строку из 10 символов. На четных позициях должны находиться четные цифры, на нечетных позициях - буквы.
5. Дана строка. Показать номера символов, совпадающих с последним символом строки.
6. Дана строка. Показать третий, шестой, девятый и так далее символы.
7. Дана строка. Определите общее количество символов '+' и '-' в ней. А так же сколько таких символов, после которых следует цифра ноль.
8. Дана строка. Определите, какой символ в ней встречается раньше: 'x' или 'w'. Если какого-то из символов нет, вывести сообщение об этом.

9. Даны две строки. Вывести большую по длине строку столько раз, на сколько символов отличаются строки.

10. Дана строка. Если она начинается на 'abc', то заменить их на 'www', иначе добавить в конец строки 'zzz'.

11. Дана строка. Если ее длина больше 10, то оставить в строке только первые 6 символов, иначе дополнить строку символами 'o' до длины 12.

12. Дана строка. Разделить строку на фрагменты по три подряд идущих символа. В каждом фрагменте средний символ заменить на случайный символ, не совпадающий ни с одним из символов этого фрагмента. Показать фрагменты, упорядоченные по алфавиту.

13. Дана строка. Заменить каждый четный символ или на 'a', если символ не равен 'a' или 'b', или на 'c' в противном случае.

14. В данной строке найти количество цифр.

15. Дана строка. Определить, содержит ли строка только символы 'a', 'b', 'c' или нет.

16. Замените в строке все вхождения 'word' на 'letter'.

17. Удалите в строке все буквы 'x'. за которыми следует 'abc'.

18. Удалите в строке все 'abc', за которыми следует цифра.

19. Найдите количество вхождения 'aba' в строку.

20. Удалить в строке все лишние пробелы, то есть серии подряд идущих пробелов заменить на одиночные пробелы. Крайние пробелы в строке удалить.

21. Даны две строки. Удалить в первой строке первое вхождение второй строки.

22. Дана строка. Определите, является ли она действительным числом.

23. Строка состоит из слов, разделенных одним или несколькими пробелами. Найдите слово наибольшей длины.

24. В строке записано десятичное число. Запишите данное число римскими цифрами.

25. Дан email в строке. Определить, является ли он корректным (наличие символа '@' и точки, наличие не менее двух символов после последней точки и т.д.).

26. Написать функцию генерации email.

27. Дано натуральное число. Получить строку, в которой тройки цифр этого числа разделены пробелом, начиная с правого конца. Например, число 1234567 преобразуется в 1 234 567.

28. Вывести слова, в которых заменить каждую большую букву одноименной малой; удалить все символы, не являющиеся буквами или цифрами; вывести в алфавитном порядке все гласные буквы, входящие в каждое слово строки.

29. Вывести текст, составленный из первых букв всех слов; все слова, содержащие больше одной буквы «s».

30. Вывести в алфавитном порядке все слова, содержащие наибольшее количество гласных букв; найти все слова, в которые буква «a» входит не менее двух раз.

31. Дана строка, которая содержит натуральные числа, знаки четырех арифметических действий (сложение, вычитание, умножение, деление) и скобки. Вычислите значение выражения.

32. Дана строка, представляющая из себя арифметическое выражение, состоящее из чисел, скобок и арифметических операций. Проверьте данное выражение на правильность расстановки скобок.

33. Дана строка, содержащая полное имя файла (например, 'c:\WebServers\home\testsite\www\myfile.txt'). Выделите из этой строки имя файла без расширения.

34. Дана строка. Если символы в ней упорядочены по алфавиту, то вывести 'yes', иначе вывести первый символ, нарушающий алфавитный порядок.

35. Дана строка, содержащая буквы и скобки '(', ')', '[', ']', '{', '}'. Если скобки расставлены правильно (то есть каждой открывающей скобки соответствует закрывающая того же вида), то вывести 'yes', иначе вывести номер первой ошибочной скобки или -1, если закрывающая скобка отсутствует.

36. Даны две строки. Выделить из каждой строки наибольшей длины подстроки, состоящие только из цифр, и объединить эти подстроки в одну новую строку.

37. Даны три строки. Заменить все вхождения второй строки в первую на третью строку.

38. Дан текст. Найдите в нем все числа, окруженные пробелами, и добавьте перед ними '<' и после них '>'.

39. Дан текст. Некоторые его фрагменты выделены группами символов ##. Заменить выделение группами символов '<' и '>'). Пример: 'Это ##тестовый пример## для задачи ##на## строки' преобразуется в 'Это <тестовый> пример для задачи <на> строки'

40. Дан текст и список слов. Найти в тексте все слова, каждое из которых отличается от некоторого слова из списка одной буквой, и исправить такие слова на слова из списка.

Контрольные вопросы

1. Какие характерные особенности типа string в языке C#?
2. Способы создания экземпляра s типа string
3. Как в переменную типа string занести значение строки?
4. Как определить, равны ли две строки типа string между собой?
5. Как сравнить две строки типа string в лексикографическом порядке?
6. Как соединить две строки типа string?
7. Как скопировать одну строку типа string в другую?
8. Вставка подстроки начиная из заданного индекса.
9. Как определить длину строки типа string?
10. Удаление заданного количества символов из строки.