

ПЛАТФОРМА MICROSOFT.NET И ЯЗЫК ПРОГРАММИРОВАНИЯ

C#

Урок №2

Массивы и строки. Перечисления

Содержание

1. Массивы	4
Одномерные массивы	4
Многомерные массивы	7
Зубчатые массивы	12
Использование цикла <code>foreach</code>	14
2. Строки	18
Создание строки	18
Операции со строками	20
Особенности использования строк	30
3. Использование аргументов командной строки ..	35
4. Перечисления (<code>enum</code>)	37
Понятие перечисления	37

Синтаксис объявления перечисления	37
Необходимость и особенности применения перечисления	39
Установка базового типа перечисления.....	42
Использование методов для перечислений	43
6. Домашнее задание	47

1. Массивы

Одномерные массивы

Массив — это ряд связанных между собой элементов *данных одинакового типа*.

Программа может обращаться к любому элементу массива путем указания имени массива, за которым в квадратных скобках следует целочисленное значение, указывающее на местоположение элемента в рамках массива, называемое индексом элемента.

Согласно общезыковой спецификации (CLS), нумерация элементов в массиве должна начинаться с нуля. Если следовать этому правилу, тогда методы, написанные на C#, смогут передать ссылку на созданный массив коду, написанному на другом языке, например, на Microsoft Visual Basic .NET. Кроме того, Microsoft постаралась оптимизировать работу массивов с начальным нулевым индексом, поскольку они получили очень большое распространение. Тем не менее, в CLR допускаются и иные варианты индексации массивов, хотя это не приветствуется.

Синтаксис объявления одномерного массива следующий

```
Тип_элементов_массива [] имя_массива;
```

Примеры объявления одномерных массивов:

```
int[] myArray;           // Объявление ссылки на массив
                           //целых чисел
string [] myStrings;     // Объявление ссылки на массив строк
```

Все массивы в C# унаследованы от класса `System.Array`. Это наследование означает, что все массивы являются объектами. С одной стороны это дает ряд преимуществ, с другой — имеет ряд недостатков. К преимуществам можно отнести: полученный в наследство от класса `System.Array` немалый набор методов по работе с массивами, контроль выхода за границы массива и др.; к недостаткам — некоторое снижение быстродействия при работе с массивом вследствие того, что он размещается в «куче».

Учитывая, что массивы — это ссылочные типы, в приведенном выше примере создаются две пустые ссылки. Для дальнейшей работы необходимо выделить память под эти ссылки и для этого используется оператор `new`.

```
myArray = new int[10];           //Выделяем память под
                                //массив на 10 чисел
myStrings = new string[50];      //Выделяем память под
                                //массив на 50 строк
```

После выделения памяти инициализация элементов производится их значениями по умолчанию: значения целых типов устанавливаются в «0», вещественных типов — в «0.0», логического типа — в «*false*», ссылочных типов — в «*null*».

Есть также возможность проинициализировать массив нужными значениями при объявлении:

```
int[] myArray1 = new int[10] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
//int[] myArray2 = new int[5]{ 2, 14, 54, 8 }; // Error
int[] myArray3 = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int[] myArray4 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

В первом случае, если количество элементов в списке инициализации окажется больше или меньше чем заказано — то компилятор выдаст сообщение об ошибке (Рисунок 1.1). Во втором и третьем случае размер массива вычисляется из количества элементов в списке инициализации.

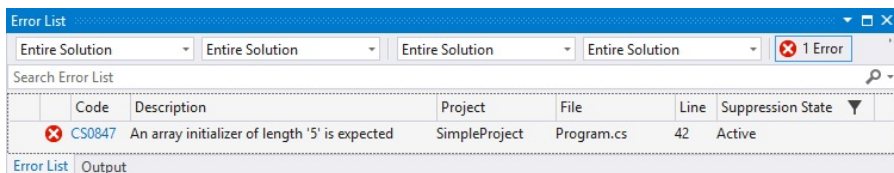


Рисунок 1.1. Ошибка инициализации массива

Инициализация ссылки на массив без использования оператора `new` приведет к ошибке на этапе компиляции (Рисунок 1.2).

```
int[] myArray;
//myArray = { 2, 14, 54, 8, 11 }; // Error
myArray = new int[] { 2, 14, 54, 8, 11 }; // OK
```

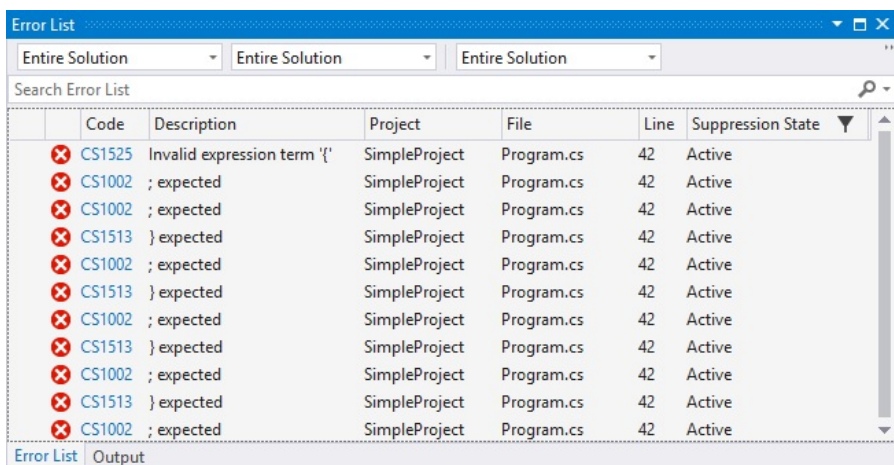


Рисунок 1.2. Ошибка на этапе компиляции

Обращение к элементам одномерного массива осуществляется через указание порядкового номера элемента — индекса:

```
myArray [2] = 53; // Присваивание значения третьему
                  //элементу массива
Console.WriteLine(myArray [2]); // Вывод на экран
                              //значения третьего элемента массива
```

Так как все массивы являются объектами, то в них помимо данных-значений хранится и дополнительная информация (размер, нижняя граница, тип массива, количество измерений, количество элементов в каждом измерении и др.). Размещение массива в памяти можно схематически представить так:

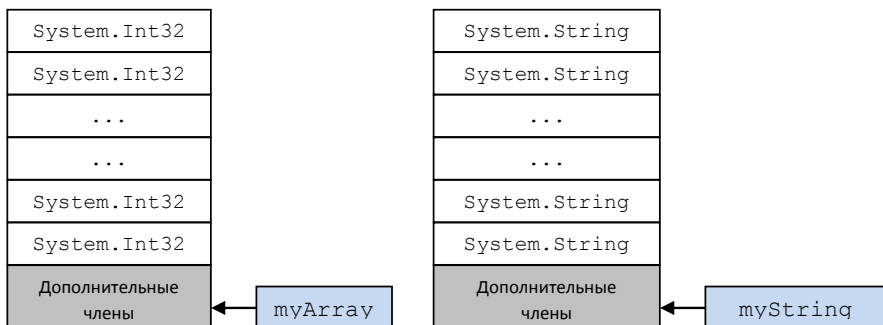


Рисунок 1.3. Схема размещения массива в памяти

Многомерные массивы

Массивы могут быть не только одномерными, но и многомерными, то есть иметь несколько измерений. Синтаксис многомерного массива приведен ниже.

```
Тип_элементов_массива [, ..., ] имя_массива;
```

Примеры объявления и инициализации двумерных массивов:

```
float[, ] myArr = new float[2, 3]; //Объявление
                                   //двумерного массива
                                   //вещественных чисел
//Объявление и инициализация двумерных
//массивов целых чисел
int[, ] myArr1 = new int[2, 3] {{ 1, 2, 3 }, { 3, 4, 6 }};
int[, ] myArr2 = new int[, ] {{ 1, 2, 3 }, { 3, 4, 6 }};
int[, ] myArr3 = {{ 1, 2, 3 }, { 3, 4, 6 }};
```

Доступ к элементам двумерного массива осуществляется через указание строки/столбца следующим образом:

```
myArr1[1, 2] = 100; // Занесение значения в третью
                   // колонку второй строки
Console.WriteLine(myArr1[0, 0]);
                   // Вывод 1 (элемент первой
                   // колонки в первой строке)
```

Примеры объявления и инициализации трехмерных массивов приведены ниже:

```
int[, , ] array1 = new int[3, 4, 2]; //трехмерный массив
                                   //целых чисел
//Объявление и инициализация трехмерных массивов целых чисел
int[, , ] array2 = new int[, , ] {{{1, 2, 3}, {4, 5, 6}},
                                   {{7, 8, 9}, {10, 11, 12}}};
int[, , ] array3 = new int[2, 2, 3] {{{1, 2, 3}, {4, 5, 6}},
                                   {{7, 8, 9}, {10, 11, 12}}};
```

Поскольку все массивы унаследованы от класса `System.Array`, то при работе с ними можно использовать методы данного класса. Давайте рассмотрим некоторые из методов класса `System.Array`:

- **GetLength** возвращает количество элементов массива по заданному измерению.
- **GetLowerBound** и **GetUpperBound** возвращают соответственно нижнюю и верхнюю границы массива по заданному измерению (например, если есть одномерный массив на 5 элементов, то нижняя граница будет «0», верхняя — «4»).
- **CopyTo** копирует все элементы одного одномерного массива в другой, начиная с заданной позиции.
- **Clone** производит поверхностное копирование массива. Копия возвращается в виде массива **System.Object[]**.
- Статический метод **BinarySearch** производит бинарный поиск значения в массиве (в диапазоне массива).
- Статический метод **Clear** присваивает значения по умолчанию типа элемента каждого элемента в массиве.
- Статический метод **IndexOf** — возвращает индекс первого вхождения искомого элемента в массиве, в случае неудачи — возвращает «-1». Поиск производится от начала массива.
- Статический метод **LastIndexOf** — возвращает индекс первого вхождения искомого элемента в массиве. Поиск производится с конца массива, в случае неудачи — возвращает «-1».
- Статический метод **Resize** изменяет размер массива.
- Статический метод **Reverse** — реверсирует массив (диапазон массива).
- Статический метод **Sort** — сортирует массив (диапазон массива).

Также присутствуют методы расширения:

- **Sum** — суммирует элементы массива.
- **Average** — подсчитывает среднее арифметическое элементов массива.
- **Contains** — возвращает истину, если заданный элемент присутствует в массиве.
- **Max** — возвращает максимальный элемент массива.
- **Min** — возвращает минимальный элемент массива.

И напоследок пара свойств:

- Свойство **Length** — возвращает длину массива.
- Свойство **Rank** — возвращает количество измерений массива.

Для закрепления полученной информации рассмотрим следующий пример:

```
static void Main(string[] args)
{
    int[] myArr1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    PrintArr("Массив myArr1", myArr1);
    int[] tempArr = (int[])myArr1.Clone();
    Array.Reverse(myArr1, 3, 4);
    PrintArr("Массив myArr1 после реверсирования", myArr1);
    myArr1 = tempArr;
    PrintArr("Массив myArr1 после восстановления", myArr1);

    int[] myArr2 = new int[20];
    PrintArr("Массив myArr2 до копирования", myArr2);
    myArr1.CopyTo(myArr2, 5);
    PrintArr("Массив myArr2 после копирования", myArr2);
}
```

```

Array.Clear(myArr2, 0, myArr2.GetLength(0));
PrintArr("Массив myArr2 после очистки: ", myArr2);
Array.Resize(ref myArr2, 10);
PrintArr("Массив myArr2 после изменения размера:", myArr2);
myArr2 = new[] { 1, 5, 3, 2, 8, 9, 6, 10, 7, 4 };
PrintArr("Несортированный массив myArr2: ", myArr2);
Array.Sort(myArr2);
PrintArr("Массив myArr2 после сортировки: ", myArr2);
Console.WriteLine("Число 5 находится в массиве на " +
    Array.BinarySearch(myArr2, 5) +
    " позиции");
Console.WriteLine("Максимальный элемент в массиве
    myArr2: " + myArr2.Max());
Console.WriteLine("Минимальный элемент в массиве
    myArr2: " + myArr2.Min());
Console.WriteLine("Среднее арифметическое элементов
    myArr2: " + myArr2.Average());

int[, ] myArr3 = { { 1, 2, 3 }, { 4, 5, 6 } };
Console.WriteLine("Количество измерений массива
    myArr3: " + myArr3.Rank);
}

static void PrintArr(string text, int[] arr)
{
    Console.Write(text + ": ");
    for (int i = 0; i < arr.Length; ++i)
        Console.Write(arr[i] + " ");
    Console.WriteLine();
}

```

Результат выполнения кода представлен на рисунке 1.4:

```

C:\WINDOWS\system32\cmd.exe
Массив myArr1: 1 2 3 4 5 6 7 8 9 10
Массив myArr1 после реверсирования: 1 2 3 7 6 5 4 8 9 10
Массив myArr1 после восстановления: 1 2 3 4 5 6 7 8 9 10
Массив myArr2 до копирования: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Массив myArr2 после копирования: 0 0 0 0 0 1 2 3 4 5 6 7 8 9 10 0 0 0 0 0
Массив myArr2 после чистки: : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Массив myArr2 после изменения размера: : 0 0 0 0 0 0 0 0 0 0
Несортированный массив myArr2: : 1 5 3 2 8 9 6 10 7 4
Массив myArr2 после сортировки: : 1 2 3 4 5 6 7 8 9 10
Число 5 находится в массиве на 4 позиции
Максимальный элемент в массиве myArr2: 10
Минимальный элемент в массиве myArr2: 1
Среднее арифметическое элементов myArr2: 5,5
Количество измерений массива myArr3: 2
Press any key to continue . . .

```

Рисунок 1.4. Пример работы с массивами

В данном примере иллюстрируется работа с методами массива, а также передача массива в метод (в нашем случае — это метод **PrintArr**).

Зубчатые массивы

Кроме одномерных и многомерных массивов C# также поддерживает зубчатые (*jagged*) массивы. Синтаксис объявления такого массива выглядит так:

```
Тип_элементов_массива [][] имя_массива;
```

Зубчатый массив представляет собой массив массивов, то есть в каждой ячейке такого массива располагается одномерный массив.

```

int[][] myArr = new int[2][]; //Создание внешнего массива
                             //на две ячейки
myArr[0] = new int[] {1, 2}; //Создание одномерного
                             //массива в первой ячейке
myArr[1] = new int[] {3, 4, 5, 6}; //Создание одномерного
                             //массива во второй ячейке

```

В результате получаем следующий массив:

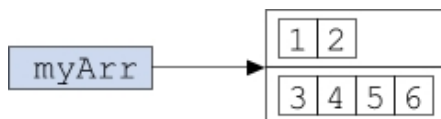


Рисунок 1.5. Схема зубчатого массива

Доступ к элементам такого массива осуществляется следующим образом:

```
Console.WriteLine(myArr[1][2]); //На экране увидим 5
```

Поскольку работа с данным видом массивов может вызывать трудности — рассмотрим пример, в котором происходит заполнение массива и вывод его на экран:

```
static void Main(string[] args)
{
    int size = 5;
    int[][] arr = new int[size][]; //объявление вложенного
                                   //массива
    for (int i = 0; i < arr.Length; i++)
    {
        arr[i] = new int[i + 1]; //создание внутренних
                                   //массивов
    }
    for (int i = 0; i < arr.Length; i++) // Length -
                                         //количество строк
    {
        for (int j = 0; j < arr[i].Length; j++) //Length -
                                                //количество элементов текущего внутреннего массива
        {
            arr[i][j] = i + j + 1; //заполнение внутренних
                                    //массивов
            Console.Write(arr[i][j] + " "); //вывод на
                                                //экран элементов
        }
    }
}
```

```

    }
    Console.WriteLine();
}
}

```

Результат работы программы (Рисунок 1.6):

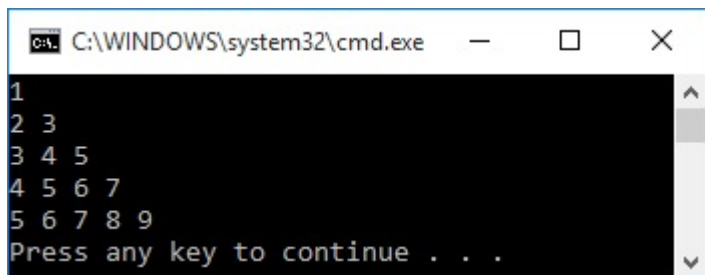


Рисунок 1.6. Пример работы с зубчатым массивом

Использование цикла `foreach`

В этом разделе мы рассмотрим использование цикла «`foreach`» при работе с массивами. Как вы помните, его предназначение — поочередный перебор элементов коллекции от начала до конца. Данный цикл удобен тем, что при работе с массивами Вам не придется вводить переменные для прохода по массиву, учитывать его длину и следить за приращением, цикл «`foreach`» все это делает сам (Рисунок 1.7).

```

static void Main(string[] args)
{
    int[] myArr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    foreach (int i in myArr)
    {

```

```

        Console.Write(i + " ");
    }
    Console.WriteLine();
}

```

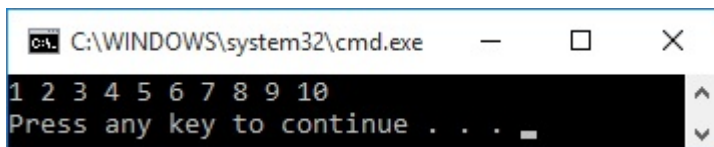


Рисунок 1.7. Работа цикла `foreach`

Единственное, что не позволяет отказаться от циклов «for» и «while» в пользу «foreach» — это то, что данный цикл работает в режиме чтения, и изменить элементы массива внутри этого цикла невозможно, это вызовет ошибку на этапе компиляции (Рисунок 1.8).

```

static void Main(string[] args)
{
    foreach (int i in myArr)
    {
        i = 23; // Error
        Console.Write(i + " ");
    }
}

```

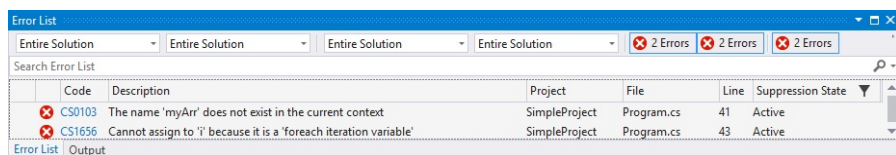


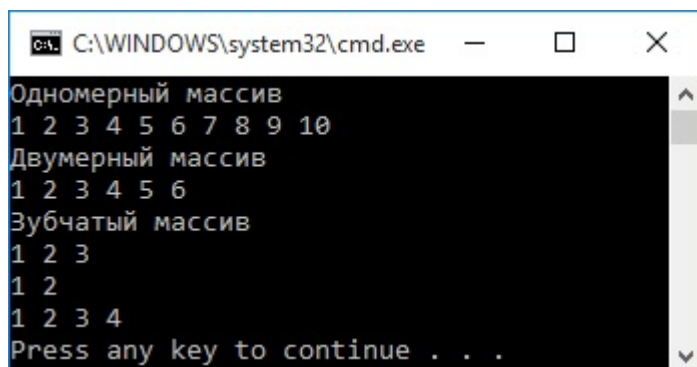
Рисунок 1.8. Ошибка: изменение элементов массива внутри цикла `foreach`

При работе с многомерными массивами цикл «foreach» не совсем удобен, потому что он выведет элементы всех измерений в одну строку.

```
static void Main(string[] args)
{
    int[] myArr1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int[,] myArr2 = { {1, 2, 3}, {4, 5, 6} };
    int[][] myArr3 = new int[3][]{new int[3]{1,2,3},
                                   new int[2]{1,2},
                                   new int[4]{1,2,3,4}};

    Console.WriteLine("Одномерный массив");
    foreach (int i in myArr1)
    {
        Console.Write(i + " ");
    }
    Console.WriteLine("\nДвумерный массив");
    foreach (int i in myArr2)
    {
        Console.Write(i + " ");
    }
    Console.WriteLine("\nЗубчатый массив");
    for (int i = 0; i < myArr3.Length; ++i)
    {
        foreach (int j in myArr3[i])
        {
            Console.Write(j + " ");
        }
        Console.WriteLine();
    }
}
```

Результат выполнения программы (Рисунок 1.9):



```
CH C:\WINDOWS\system32\cmd.exe
Одномерный массив
1 2 3 4 5 6 7 8 9 10
Двумерный массив
1 2 3 4 5 6
Зубчатый массив
1 2 3
1 2
1 2 3 4
Press any key to continue . . .
```

Рисунок 1.9. Примеры использования цикла foreach с массивами

2. Строки

Создание строки

Ссылочный тип данных `string` представляет последовательность из нуля или более символов в кодировке Unicode и является псевдонимом для класса `System.String` платформы .NET Framework.

Поскольку строки имеют тип `System.String` — все они являются объектами. Из этого следует то, что строки размещаются в «куче» и имеют богатый набор методов. Несмотря на то, что строка это ссылка — создавать ее удобно без ключевого слова `new`.

```
string имя_строки = значение;
```

Несмотря на такой простой способ создания строки, класс `System.String` имеет 8 конструкторов, с помощью которых, в основном, происходит создание строки из массива символов. Примеры использования таких конструкторов приведены ниже.

```
string str1 = "Простая строка";
char[] chrArr={'П','р','о','с','т','а','я',' ','с','т','р','о','к','а'};
string str2 = new string(chrArr);
string str3 = new string(chrArr, 8, 6);
string str4 = new string('$', 10);
Console.WriteLine("str1: " + str1);
Console.WriteLine("str2: " + str2);
Console.WriteLine("str3: " + str3);
Console.WriteLine("str4: " + str4);
```

Результат выполнения кода (Рисунок 2.1):

```

C:\WINDOWS\system32\cmd.exe
str1: Простая строка
str2: Простая строка
str3: строка
str4: $$$$$$$$$$
Press any key to continue . . .

```

Рисунок 2.1. Примеры использования конструкторов класса `System.String`

Так же как и в языке C++ — в C# существует последовательность управляющих символов. Все символы этой последовательности начинаются с символа '\'. Поэтому если мы хотим включить символы обратной косой черты, одинарной или двойной кавычки — непосредственно в строковый литерал, то понадобится указать дополнительный символ '\' перед вышеперечисленными (Рисунок 2.2).

```

string str = "\\Компьютерная академия \"ШАГ\"\\\"";
Console.WriteLine(str);

```

```

C:\WINDOWS\system32\cmd.exe
мпьютерная академия \"ШАГ\"\\
ss any key to continue . . .

```

Рисунок 2.2. Использование управляющих символов

Так как часто возникает необходимость работы с путями к файлам или папкам — было введено понятие

«буквальных» (*verbatim*) строк. Перед такими строками ставится символ '@' и все символы строкового литерала воспринимаются буквально, как они есть.

```
string strPath1 = "D:\\Student\\MyProjects\\Strings\\";  
string strPath2 = @"D:\\Student\\MyProjects\\Strings\\";  
Console.WriteLine(strPath1);  
Console.WriteLine(strPath2);
```

В результате получаем два одинаковых вывода (Рисунок 2.3):

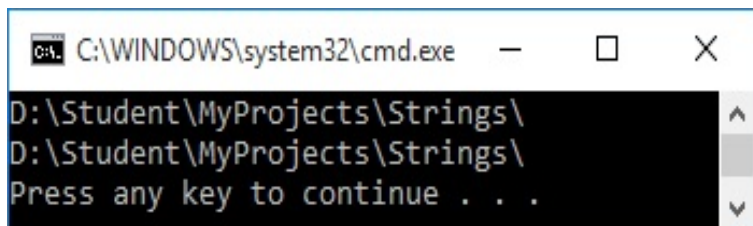


Рисунок 2.3. Использование «буквальных» строк

Операции со строками

В классе `System.String` содержится достаточное количество методов, позволяющих выполнять различные операции со строками, проведем обзор возможностей этого класса:

- Индексатор — позволяет по индексу получить символ строки. Работает только в режиме чтения.
- Свойство **Length** — возвращает длину строки.
- Метод **CopyTo** — копирует заданное количество символов в массив типа `char`.

```

static void Main(string[] args)
{
    string str = "Простая строка";
    char[] chrArr = new char[6];

    Console.WriteLine("Реверсирование строки с помощью
                      индексатора");
    for (int i = str.Length - 1; i >= 0; --i)
        Console.Write(str[i]);

    Console.WriteLine("\nКопирование строки
                      в массив символов");
    //Копируем шесть символов начиная с восьмой позиции и
    //помещаем в массив
    str.CopyTo(8, chrArr, 0, 6);
    Console.WriteLine(chrArr);
}

```

Результат выполнения (Рисунок 2.4):

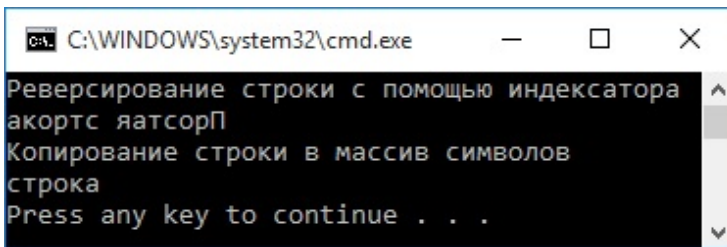


Рисунок 2.4. Пример использования операций со строками

- Метод **Equals** — сравнивает значение двух строк.
- Статический метод **Compare** — сравнивает две строки, переданные в качестве аргументов.
- Статический метод **CompareOrdinal** — сравнивает две строки, оценивая числовые значения соответствующих символов в каждой строке.

- Метод **CompareTo** — сравнивает данную строку со строкой, переданной в качестве параметра. Возвращает целое значение: меньше нуля — данная строка предшествует параметру, нуль — строки равны, больше нуля — данная строка стоит после параметра.
- Метод **StartsWith** — определяет, начинается ли текущая строка с указанной подстроки.
- Метод **EndsWith** — определяет, заканчивается ли текущая строка указанной подстрокой.

```
static void Main(string[] args)
{
    string str1 = "Простая строка";
    string str2 = "Строка";
    string str3 = "строка";
    string[] strArr = { "ШАГ", "шагаем", "бежим", "ем",
                       "Играем" };

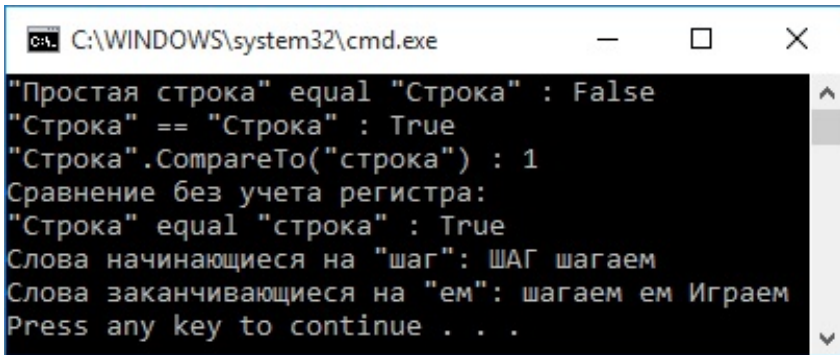
    Console.WriteLine("\"" + str1 + "\" equal \"" + str2
                      + "\" : " + str1.Equals(str2));
    Console.WriteLine("\"" + str2 + "\" == \"Строка\" : " +
                      (str2 == "Строка"));
    Console.WriteLine("\"" + str2 + "\".CompareTo(\"" +
                      str3 + "\") : " + str2.CompareTo(str3));
    Console.WriteLine("Сравнение без учета регистра:");
    Console.WriteLine("\"" + str2 + "\" equal \"" +
                      str3 + "\" : " + str2.Equals(str3,
                      StringComparison.CurrentCultureIgnoreCase));
    Console.WriteLine("Слова начинающиеся на \"шаг\": ");
    foreach (string s in strArr)
        if (s.StartsWith("шаг",
                        StringComparison.CurrentCultureIgnoreCase))
            Console.WriteLine(s + " ");
    Console.WriteLine("\nСлова заканчивающиеся на
                      \"ем\": ");
}
```

```

foreach(string s in strArr)
    if(s.EndsWith("ем",
        StringComparison.CurrentCultureIgnoreCase))
        Console.Write(s + " ");
    Console.WriteLine();
}

```

Результат выполнения (Рисунок 2.5):



```

C:\WINDOWS\system32\cmd.exe
"Простая строка" equal "Строка" : False
"Строка" == "Строка" : True
"Строка".CompareTo("строка") : 1
Сравнение без учета регистра:
"Строка" equal "строка" : True
Слова начинающиеся на "шаг": ШАГ шагаем
Слова заканчивающиеся на "ем": шагаем ем Играем
Press any key to continue . . .

```

Рисунок 2.5. Пример использования операций со строками

- Метод **IndexOf** и **LastIndexOf** — возвращает индекс первого/последнего вхождения символа/подстроки в исходной строке.
- Методы **IndexOfAny** и **LastIndexOfAny** возвращает индекс первого/ последнего вхождения любого из перечисленных символов в исходной строке.
- Метод **Substring** возвращает подстроку из текущей строки.

Все методы поиска включают перегруженные версии для поиска в заданном диапазоне с заданным способом сравнения.

```
static void Main(string[] args)
{
    string str1 = "ПолиморфизмНаследованиеИнкапсуляция";
    string str2 = "АБВГДЕЖЗИКЛМН";

    Console.WriteLine("Первое вхождение символа \'Н\': " +
        str1.IndexOf('Н'));
    Console.WriteLine("Первое вхождение подстроки
        \\'Наследование\' : " +
        str1.IndexOf('Н'));
    Console.WriteLine("Последнее вхождение символа \'И\': " +
        str1.LastIndexOf('И'));
    Console.WriteLine("Последнее вхождение любого из
        символов строки " +
        "\'АБВГДЕЖЗИКЛМН\' : " +
        str1.LastIndexOfAny(str2.ToCharArray()));
    Console.WriteLine("Подстрока начиная с 11 символа
        по 23-й : " + str1.Substring(11, 12));
}
```

Результат выполнения (Рисунок 2.6):

```
cmd. C:\WINDOWS\system32\cmd.exe
Первое вхождение символа 'Н': 11
Первое вхождение подстроки "Наследование" : 11
Последнее вхождение символа 'И': 23
Последнее вхождение любого из символов строки "АБВГДЕЖЗИКЛМН" : 23
Подстрока начиная с 11 символа по 23-й : Наследование
Press any key to continue . . .
```

Рисунок 2.6. Пример использования операций со строками

- Статический метод **Concat** осуществляет конкатенацию (объединение) строк. Удобная альтернатива данному методу — операции «+» и «+=».
- Методы **ToLower** и **ToUpper** — возвращают строку в нижнем и верхнем регистре соответственно.

- Метод **Replace** заменяет все вхождения символа/подстроки на заданный символ/подстроку.
- Метод **Contains** — проверяет, входит ли заданный символ/подстрока в исходную строку.
- Метод **Insert** — вставляет подстроку в заданную позицию исходной строки.
- Метод **Remove** — удаляет из текущей строки все вхождения указанной подстроки.
- Методы **PadLeft** и **PadRight** дополняют исходную строку заданными символами слева/справа. Если символ не указывается, то дополнение происходит символом пробела. Первый параметр указывает на количество символов в строке, до которого она должна быть дополнена.
- Метод **Split** разбивает строку на подстроки по заданным символам разделителям. Возвращает массив получившихся в результате разбиения строк. Чтобы исключить из этого массива пробельные строки — нужно использовать данный метод с параметром `StringSplitOptions.RemoveEmptyEntries`.
- Статический метод **Join** объединяет строки заданного массива в одну и чередует их с указанным символом-разделителем.
- Методы **TrimLeft** и **TrimRight** убирают пробельные (по умолчанию) или заданные символы соответственно с начала и конца строки. Метод **Trim** — делает тоже с обеих сторон строки.

И теперь пример на вышеперечисленные методы:

```
static void Main(string[] args)
{
    string str1 = "Я ";
    string str2 = "учу ";
    string str3 = "С#";
    string str4 = str1 + str2 + str3;

    Console.WriteLine("{0} + {1} + {2} = {3}", str1, str2,
        str3, str4);

    str4 = str4.Replace("учу", "изучаю");
    Console.WriteLine(str4);

    str4 = str4.Insert(2, "упорно ").ToUpper();
    Console.WriteLine(str4);

    if (str4.Contains("упорно"))
        Console.WriteLine("Учу таки упорно :)");
    else
        Console.WriteLine("Учу как могу");

    str4 = str4.PadLeft(25, '*');
    str4 = str4.PadRight(32, '*');
    Console.WriteLine(str4);
    str4 = str4.TrimStart("*".ToCharArray());
    Console.WriteLine(str4);
    string[] strArr = str4.Split(" *".ToCharArray(),
        StringSplitOptions.RemoveEmptyEntries);
    foreach (string str in strArr)
        Console.WriteLine(str);
    str4 = str4.Remove(9);
    str4 += "учусь";
    Console.WriteLine(str4);
}
```

Результаты выполнения (Рисунок 2.7):

```

C:\WINDOWS\system32\cmd.exe
Я + учу + С# = Я учу С#
Я изучаю С#
Я УПОРНО ИЗУЧАЮ С#
Учу как могу
*****Я УПОРНО ИЗУЧАЮ С#*****
Я УПОРНО ИЗУЧАЮ С#*****
Я
УПОРНО
ИЗУЧАЮ
С#
Я УПОРНО учусь
Press any key to continue . . .

```

Рисунок 2.7. Пример использования операций со строками

- Статический метод **Format** — позволяет удобно сформатировать строку. Первый параметр — это форматная строка, которая содержит текст выводимый на экран. Если в эту строку необходимо вставить значения переменных, то место вставки помечается индексом в фигурных скобках, при необходимости, там же можно указать количество символов, занимаемых вставляемым элементом и его спецификатор формата. Сами вставляемые данные указываются следующими параметрами метода. Таким образом, синтаксис использования метода **Format** следующий:

```
String.Format("Печатаемый текст {индекс,
               размер:спецификатор}", данные);
```

Спецификаторы формата:

1. «С» или «с» — для числовых данных. Выводит символ местной валюты.

2. «D» или «d» — для целочисленных данных. Выводит обычное целое число.
3. «E» или «e» — для числовых данных. Выводит число в экспоненциальной форме.
4. «F» или «f» — для числовых данных. Выводит число с фиксированной десятичной точкой.
5. «G» или «g» — для числовых данных. Выводит обычное число.
6. «N» или «n» — для числовых данных. Выводит числа в формате локальных настроек.
7. «P» или «p» — для числовых данных. Выводит числа с символом '%」.
8. «X» или «x» — для целочисленных данных. Выводит число в шестнадцатеричном формате.

```
double test1=99989.987;  
int test2 = 99999;  
  
Console.WriteLine(String.Format("c format: {0,15:C}",  
                                test1));  
Console.WriteLine(String.Format("D format: {0:D9}", test2));  
Console.WriteLine(String.Format("E format: {0:E}", test1));  
Console.WriteLine(String.Format("f format: {0:F2}", test1));  
Console.WriteLine(String.Format("G format: {0:G}", test1));  
Console.WriteLine(String.Format("N format: {0,15:N}",  
                                test2));  
Console.WriteLine(String.Format("P format: {0:P}", test1));  
Console.WriteLine(String.Format("X format: {0:X}", test2));  
Console.WriteLine(String.Format("x format: {0:x}", test2));
```

Результаты выполнения (Рисунок 2.8):

```

C:\WINDOWS\system32\cmd.exe
c format: 99 989,99?
D format: 000099999
E format: 9,998999E+004
f format: 99989,99
G format: 99989,987
N format: 99 999,00
P format: 9 998 998,70%
X format: 1869F
x format: 1869f
Press any key to continue . . .

```

Рисунок 2.8. Пример использования метода Format

В версии C# 6.0 появилась новое понятие — **интерполированные строки**. На самом деле это возможность формирования строки без использования метода `Format`, но с аналогичным результатом. Для этого перед строкой следует поставить символ '\$', а в самой строке в фигурных скобках указать переменную либо значение, которое необходимо вставить в строку. Также появилась возможность использования в строке тернарного оператора. Примеры программного кода и результаты (Рисунок 2.9) представлены ниже.

```

int number1 = 56, number2=45;
Console.WriteLine(String.Format("Число №1 равно {0}. Число
                                №2 равно {1}.", number1,number2));
Console.WriteLine($"Число №1 равно {number1}. Число №2
                                равно {number2}.");
// использование тернарного оператора
Console.WriteLine($"Число №1 {(number1 > number2 ?
                                "больше": "меньше")} числа №2");

```

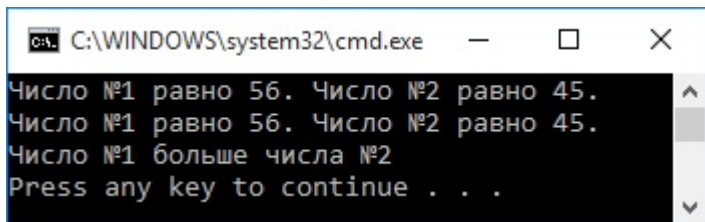
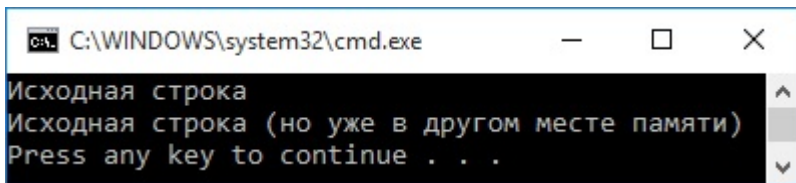


Рисунок 2.9. Примеры использования интерполированных строк

Особенности использования строк

При работе со строками нужно учитывать тот факт, что в C# строки неизменны. То есть, невозможно внести в строку любые изменения не пересоздав ее. Но беспокоиться по этому поводу не стоит — строка создается и уничтожается автоматически, вам лишь нужно принять ссылку на нее и продолжать работать. При этом нужно понимать, что ссылочные переменные типа `string` могут менять объекты, на которые они ссылаются. А содержимое созданного `string`-объекта изменить уже невозможно. Пример и результат (Рисунок 2.10) ниже.

```
static void Main(string[] args)
{
    string str1 = "Исходная строка";
    Console.WriteLine(str1);
    str1 += " (но уже в другом месте памяти)";
    Console.WriteLine(str1);
}
```

Рисунок 2.10. Особенности типа `string`

В приведенном выше примере сначала создается строка на 15 символов. Но при попытке добавить к ней другую строку — выделяется новая память под строку в 46 символов, в которую копируется содержимое обоих исходных. Старая строка при этом попадает под работу «сборщика мусора» (будет рассматриваться в соответствующем уроке).

Такая особенность строк приводит к тому, что при интенсивном изменении строки затрачивается много ресурсов (как памяти, так и «сборщика мусора» из-за чего может падать быстродействие).

Для того чтобы избежать потерь производительности, был создан класс **StringBuilder**. Данный класс имеет менее обширный набор методов по сравнению с классом `String`, но при этом мы работаем с объектом, расположенным в одном и том же месте в памяти. Память перераспределяется только тогда, когда в объекте типа **StringBuilder** не хватает места для произведенных изменений. При этом максимальное число знаков, которое может содержаться в памяти, увеличивается вдвое.

Кратко рассмотрим методы класса **StringBuilder**.

- **Append** — добавляет к исходной строке данные любого из стандартных типов.
- **AppendFormat** — добавляет к исходной строке строку, сформированную в соответствии со спецификаторами формата.
- **Insert** — вставляет данные любого из стандартных типов в исходную строку.

- **Remove** — удаляет из исходной строки диапазон символов.
- **Replace** — заменяет символ/подстроку в исходной строке на указанный символ/подстроку.
- **CopyTo** — копирует символы исходной строки в массив типа `char`.
- **ToString** — преобразовывает объект **StringBuilder** в **String**.

Так же в классе **StringBuilder** существуют следующие свойства:

- **Length** — возвращает количество символов, находящихся в строке в данный момент.
- **Capacity** — возвращает или устанавливает количество символов, которое может быть помещено в строку без дополнительного выделения памяти.
- **MaxCapacity** возвращает максимальную вместимость строки.

Класс `StringBuilder` имеет большое преимущество над классом `String` при интенсивной работе со строками, поэтому именно его рекомендуется применять в таких ситуациях. В остальных случаях более удобным является класс `String`.

Пример и результат (Рисунок 2.11) работы с классом `StringBuilder` представлены ниже.

```
StringBuilder sb = new StringBuilder();  
  
//sb = "hello"; Error  
  
sb.Append("hello"); // добавление строки к существующей
```



```
sb.AppendLine(); // добавление пустой строки к существующей
sb.AppendLine();
sb.Append("world");

Console.WriteLine("\n\tИсходная строка");
Console.WriteLine(sb);
Console.WriteLine("Максимальное количество символов " +
                  sb.Capacity);
Console.WriteLine("Длина текущего объекта " + sb.Length);

Console.WriteLine("\n\tВставка строки");
sb.Insert(7, "abracadabra"); //вставка строки в заданную
                             //позицию
Console.WriteLine(sb);
Console.WriteLine("Максимальное количество символов " +
                  sb.Capacity);
Console.WriteLine("Длина текущего объекта " + sb.Length);

Console.WriteLine("\n\tЗамена символов 'a' на 'z'");
sb.Replace('a', 'z'); // замена символов строки
Console.WriteLine(sb);

Console.WriteLine("\n\tУдаление 10 символов начиная с 3");
sb.Remove(3, 10); // удаление символов из строки
Console.WriteLine(sb);
Console.WriteLine("Максимальное количество символов " +
                  sb.Capacity);
Console.WriteLine("Длина текущего объекта " + sb.Length);
```

```

C:\WINDOWS\system32\cmd.exe
Исходная строка
hello
world
Максимальное количество символов 16
Длина текущего объекта 14

Вставка строки
hello
abracadabra
world
Максимальное количество символов 27
Длина текущего объекта 25

Замена символов 'a' на 'z'
hello
zbrzccdzbrz
world

Удаление 10 символов начиная с 3
heldzbrz
world
Максимальное количество символов 19
Длина текущего объекта 15
Press any key to continue . . .

```

Рисунок 2.11. Результаты работы с классом `StringBuilder`

Следует заметить, что инициализация класса `StringBuilder` строкой приведет к ошибке на этапе компиляции (Рисунок 2.12).

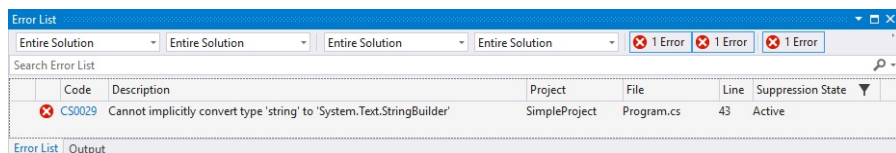


Рисунок 2.12. Ошибка инициализации класса `StringBuilder`

3. Использование аргументов командной строки

Как и в других языках программирования, в С# программу могут передаваться параметры командной строки. Для работы с параметрами командной строки существует единственный параметр в методе `Main` — `args` типа `string[]`. Этот параметр содержит передаваемые в программу аргументы командной строки.

В С++ нулевым параметром командной строки передавалось полное имя исполняемого модуля (т. е. имя файла), в С# нулевой параметр передает первый параметр командной строки. Другими словами, массив `args` содержит только параметры командной строки.

Для того чтобы с командной строкой было удобней работать во время отладки — нужные параметры можно ввести во время проектирования (чтоб запускать программу через IDE). Для этого нужно зайти в свойства проекта и выбрав вкладку «Debug» ввести необходимые параметры в окно «Command line arguments», что проиллюстрировано на рисунке 3.1.

Для получения полного пути исполняемого файла консольной программы предназначен объект `System.Environment.CommandLine`.

Рассмотрим пример и результаты (Рисунок 3.2) вывода параметров командной строки:

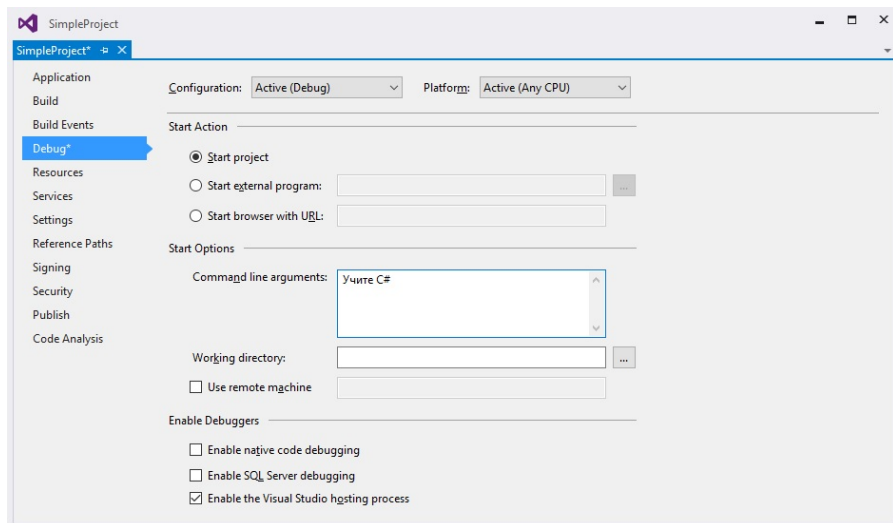


Рисунок 3.1. Внесение параметров командной строки через IDE

```
static void Main(string[] args)
{
    foreach (string item in args)
    {
        Console.WriteLine(item);
    }
}
```

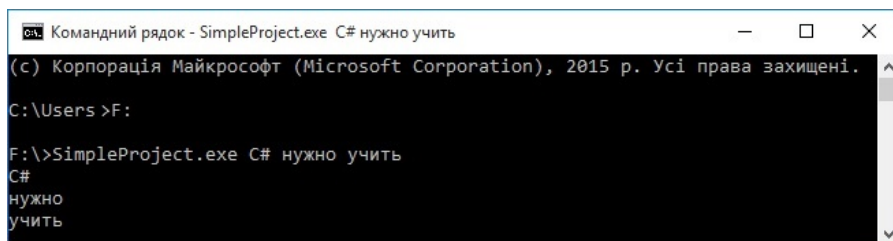


Рисунок 3.2. Вывод параметров командной строки

4. Перечисления (enum)

Понятие перечисления

Перечисление (*enumeration*) — это непустой список именованных констант. Он задает все значения, которые может принимать переменная данного типа. Перечисления являются классом и наследуются от базового класса `System.Enum`.

Синтаксис объявления перечисления

Для объявления перечисления используется ключевое слово `enum`, указывается имя перечисления и в фигурных скобках перечисляются имена констант:

```
enum EnumName {elem1, elem2, elem3, elem4}
```

Примером может быть перечисление дней недели:

```
enum DayOfWeek
{
    Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
    Sunday
}
```

Следующий пример перечисления описывает возможные типы грузовых автомобилей:

```
enum TransportType
{
    Semitrailer, Coupling, Refrigerator, OpenSideTruck, Tank
}
```

Обращение к элементу перечисления осуществляется указыванием имени класса перечисления и через точку имени конкретного элемента перечисления.

```
DayOfWeek day = DayOfWeek.Monday;
```

Константы перечисления имеют целочисленный тип `int`. По умолчанию, первой константе присваивается значение 0, а значение каждой следующей константы увеличивается на единицу. В приведенном примере перечисления `DayOfWeek` значения констант будут следующие: Monday = 0, Tuesday = 1, Wednesday = 2, Thursday = 3, Friday = 4, Saturday = 5, Sunday = 6. С переменными типов перечислений можно осуществлять арифметические операции. Приведем пример метода, возвращающего следующий день недели:

```
public DayOfWeek NextDay(DayOfWeek day)
{
    return (day < DayOfWeek.Sunday) ? ++day : DayOfWeek.Monday;
}
```

Также можно присвоить значение константе явно. Например, следующее перечисление описывает размер скидки для разных типов клиентов:

```
enum Discount
{
    Default, Incentive = 2, Patron = 5, VIP = 15
}
```

Необходимость и особенности применения перечисления

Перечисления позволяют сделать процесс программирования более удобным и быстрым, помогают избавиться от путаницы при присвоении переменным значений. Можно выделить следующие полезные функции перечислений:

- перечисления гарантируют, что переменным будут присваиваться только разрешенные, ожидаемые значения. Если вы попытаетесь присвоить экземпляру перечисления значение, которое не входит в список допустимых значений, то компилятор выдаст ошибку;
- перечисления делают код более понятным, так как мы обращаемся не просто к числам, а к осмысленным именам;
- перечисления позволяют программисту сэкономить время. Когда вы захотите присвоить экземпляру перечисляемого типа какое-то значения, то интегрированная в Visual Studio среда IntelliSense отобразит список всех допустимых значений;
- как уже упоминалось, перечисления наследуются от базового класса `System.Enum`, что позволяет вызывать для них ряд полезных методов (более подробно это будет рассмотрено ниже).

В основном, перечисления используются, когда необходимо осуществить определенные действия, исходя из совпадения значения перечисления с возможными значениями с помощью оператора `switch`.

Рассмотрим пример, в котором пользователь выбирает из списка наименование товара, а программа определя-

ет каким транспортом необходимо его перевозить. Код программы приведен ниже:

```
using System;
namespace SimpleProject
{
    enum CommodityType //тип товара
    {
        FrozenFood, Food, DomesticChemistry,
        BuildingMaterials, Petrol
    }

    enum TransportType //тип транспорта
    {
        Semitrailer, Coupling, Refrigerator, OpenSideTruck,
        FuelTruck
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Введите число от 1 до 5");
            int number = Int32.Parse(Console.ReadLine());
            if (number > 0 && number < 6)
            {
                CommodityType commodity = (CommodityType)
                Enum.GetValues(typeof(CommodityType)).
                GetValue(number - 1);

                TransportType transport = TransportType.
                Semitrailer;

                switch (commodity)
                {
                    case CommodityType.FrozenFood:
                        transport = TransportType.Refrigerator;
                        break;
                    case CommodityType.Food:
```



```

        transport = TransportType.Semitrailer;
        break;
    case CommodityType.DomesticChemistry:
        transport = TransportType.Coupling;
        break;
    case CommodityType.BuildingMaterials:
        transport = TransportType.
            OpenSideTruck;
        break;
    case CommodityType.Petrol:
        transport = TransportType.FuelTruck;
        break;
    }
    Console.WriteLine($"Для товара -
    {commodity} необходим транспорт -
    {transport}.");
}
else
{
    Console.WriteLine("Ошибка ввода");
}
}
}
}

```

Перечисления `CommodityType` и `TransportType` описывают типы товаров и типы грузовиков соответственно. На основании введенной пользователем информации и после ее проверки, мы получаем значение из перечисления `CommodityType`. Для этого используем метод `GetValues` базового класса `System.Enum`, который возвращает класс `System.Array`, метод `GetValue` которого возвращает значение объекта по индексу. После этого сравниваем типы товаров и присваиваем соответствующий тип транспорта. Один из вариантов работы программы представлен на рисунке 4.1:

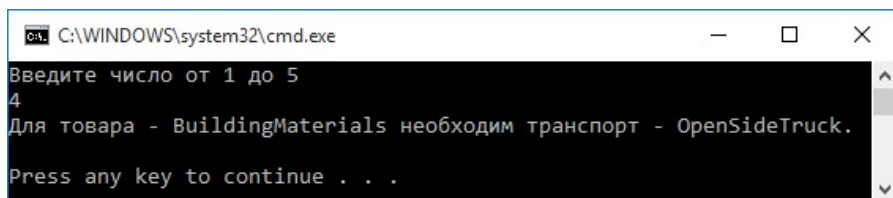


Рисунок 4.1. Вариант работы программы
с использованием перечислений

Установка базового типа перечисления

Под базовым типом понимается тип констант перечисления. Как уже упоминалось, по умолчанию перечисления основываются на типе `int`. Но можно создать перечисление на основе любого из целочисленных типов: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`. Для этого при объявлении перечисления после его имени указывается через двоеточие нужный тип:

```
enum NameEnum [:базовыйТип] {EnumList}
```

Для примера приведем перечисление, описывающее среднюю дистанцию планет солнечной системы от солнца в километрах:

```
enum DistanceSun : ulong
{
    Sun = 0,
    Mercury = 57900000,
    Venus = 108200000,
    Earth = 149600000,
    Mars = 227900000,
    Jupiter = 778300000,
    Saturn = 142700000,
```

```

Uranus = 2870000000,
Neptune = 4496000000,
Pluto = 5946000000
}

```

Использование методов для перечислений

Перечисления являются классом, и наследуется от базового класса `System.Enum`. Это значит, что для них можно использовать методы данного класса, то есть методы сравнения значений перечисления, методы преобразования значений перечисления в строковое представление, методы преобразования строкового представления значения в перечисление и другие. Вот некоторые из них:

- **CompareTo** — сравнивает текущий экземпляр с заданным объектом и возвращает: значение меньше 0, если значение текущего экземпляра меньше заданного; 0, если значения равны; значение больше 0, если значение текущего экземпляра больше заданного.
- **GetName** — выводит имя константы в указанном перечислении, имеющем заданное значение.
- Статический метод **GetNames** — выводит массив имен констант в указанном перечислении.
- Статический метод **GetValues** — выводит массив значений констант в указанном перечислении.
- Статический метод **IsDefined** — возвращает признак наличия константы с указанным значением в заданном перечислении. Возвращает `true`, если константа присутствует, иначе — `false`.

- Статический метод **Parse** — преобразует строковое представление имени или числового значения одной или нескольких перечисляемых констант в эквивалентный перечисляемый объект.
- **ToString** — Преобразует значение этого экземпляра в эквивалентное ему строковое представление.

Пример использования методов класса `System.Enum` при работе с перечислениями представлен ниже:

```
using System;

namespace SimpleProject
{
    enum DistanceSun : ulong
    {
        Sun = 0, Mercury = 57900000, Venus = 108200000,
        Earth = 149600000,
        Mars = 227900000, Jupiter = 778300000,
        Saturn = 142700000,
        Uranus = 287000000, Neptune = 449600000,
        Pluto = 594600000
    }

    class Program
    {
        static void Main(string[] args)
        {
            string moon = "Moon";

            //проверка наличия константы в заданном перечислении
            if (!Enum.IsDefined(typeof(DistanceSun), moon))
            {
                Console.WriteLine($"{moon} нет в
                                    перечислении DistanceSun.");
            }
        }
    }
}
```

```

Console.WriteLine("\n\tФорматированный вывод
всех значений констант указанного перечисления.");
foreach (DistanceSun item in Enum.
    GetValues(typeof(DistanceSun)))
{
    Console.WriteLine("{0,-10} {1,-10} {2,20}",
        //вывод в виде строки с именем константы
        Enum.Format(typeof(DistanceSun), item, "G"),
        //вывод в виде десятичного значения
        Enum.Format(typeof(DistanceSun), item, "D"),
        //вывод в виде 16-ричного значения
        Enum.Format(typeof(DistanceSun), item, "X"));
}
Console.WriteLine("\n\tВсе значения констант
указанного перечисления.");
foreach (string str in Enum.
    GetNames(typeof(DistanceSun)))
{
    Console.WriteLine(str);
}

ulong number = 227900000;
Console.WriteLine($" \n\tИмя константы со значением
{number} из указанного перечисления.\n");
Console.WriteLine(Enum.GetName(typeof(DistanceSun),
    number));
}
}

```

Результат выполнения кода (Рисунок 4.2):

```

C:\WINDOWS\system32\cmd.exe

Значения Moon нет в перечислении DistanceSun.

Форматированный вывод всех значений констант указанного перечисления.
Sun          0          0000000000000000
Mercury      57900000    0000000003737BE0
Venus       108200000    0000000006730040
Earth       149600000    0000000008EAB700
Mars        227900000    000000000D957A60
Saturn      142700000    000000000550E4AC0
Uranus      287000000    000000000AB10B980
Neptune     449600000    0000000010BF88400
Pluto       594600000    0000000016268C280
Jupiter     778300000    000000001CFE727C0

Все значения констант указанного перечисления.
Sun
Mercury
Venus
Earth
Mars
Saturn
Uranus
Neptune
Pluto
Jupiter

Имя константы со значением 227900000 из указанного перечисления.
Mars
Press any key to continue . . .

```

Рисунок 4.2. Работа программы с использованием перечислений

5. Домашнее задание

1. Сжать массив, удалив из него все 0 и, заполнить освободившиеся справа элементы значениями -1
2. Преобразовать массив так, чтобы сначала шли все отрицательные элементы, а потом положительные (0 считать положительным)
3. Написать программу, которая предлагает пользователю ввести число и считает, сколько раз это число встречается в массиве.
4. В двумерном массиве порядка M на N поменяйте местами заданные столбцы.

© Юрий Задерей.

© Компьютерная Академия «Шаг»

www.itstep.org.

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.