

Лекция 8

Структурные типы данных

- 1. Структуры. Синтаксис и применение**
- 2. Перечисления**
- 3. Операции с перечислениями**
- 4. Базовый класс `System.Enum`**

1. Структуры. Синтаксис и применение

Структура – тип данных, аналогичный классу, но имеющий ряд важных отличий от него:

- структура является *значимым*, а не ссылочным типом данных, то есть экземпляр структуры хранит значения своих элементов, а не ссылки на них, и располагается в стеке, а не в хипе;
- структура не может участвовать в иерархиях наследования, она может только реализовывать интерфейсы;
- в структуре запрещено определять конструктор по умолчанию, поскольку он определен неявно и присваивает всем ее элементам значения по умолчанию (нули соответствующего типа);
- в структуре запрещено определять деструкторы, поскольку это бессмысленно.

Отличия от классов обуславливают область применения структур: типы данных, имеющие небольшое количество полей, с которыми удобнее работать как со значениями, а не как со ссылками. Накладные расходы на динамическое выделение памяти для небольших объектов могут весьма значительно снизить быстродействие программы, поэтому их эффективнее описывать как структуры, а не как классы.

Синтаксис структуры:

```
[атрибуты] [спецификаторы] struct имя_структуры [:  
интерфейсы]  
тело структуры [;]
```

Спецификаторы структуры имеют такой же смысл, как и для класса, причем из спецификаторов доступа допускаются только **public**, **internal** и **private** (последний – только для вложенных структур).

Интерфейсы, реализуемые структурой, перечисляются через запятую.

Тело структуры может состоять из констант, полей, методов, свойств, событий, индексаторов, операций, конструкторов и вложенных типов. Правила их описания и использования аналогичны соответствующим элементам классов.

Отличия структур от классов:

- поскольку структуры не могут участвовать в иерархиях, для их элементов не могут использоваться спецификаторы **protected** и **protected internal**;
- структуры не могут быть абстрактными (**abstract**), к тому же по умолчанию они бесплодны (**sealed**);
- методы структур не могут быть абстрактными и виртуальными;
- переопределяться (то есть описываться со спецификатором **override**) могут только методы, унаследованные от базового класса **object**;
- параметр **this** интерпретируется как значение, поэтому его можно использовать для ссылок, но не для присваивания;
- при описании структуры нельзя задавать значения полей по умолчанию – это будет сделано в конструкторе по умолчанию, создаваемом автоматически (конструктор присваивает значимым полям структуры нули, а ссылочным – значение **null**).

Примечание – К статическим полям это ограничение не относится.

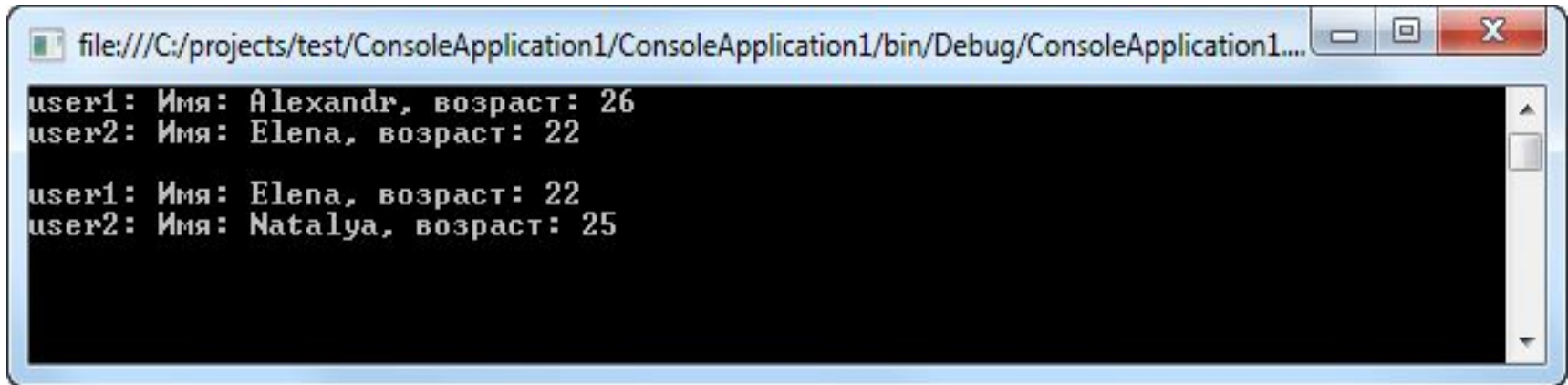
Листинг 1 – Пример 1 использования структур

```
using System;
namespace ConsoleApplication1
{
    // Создадим структуру
    struct UserInfo
    {
        public string Name;
        public byte Age;

        public UserInfo(string Name, byte Age)
        {
            this.Name = Name;
            this.Age = Age;
        }
        public void WriteUserInfo()
        {
            Console.WriteLine("Имя: {0}, возраст:
{1}", Name, Age);
        }
    }
}
```

```
class Program
{
static void Main()
    {UserInfo user1 = new UserInfo("Alexandr", 26);
      Console.Write("user1: ");
      user1.WriteUserInfo();
      UserInfo user2 = new UserInfo("Elena",22);
      Console.Write("user2: ");
      user2.WriteUserInfo();
//    Показать главное отличие структур от классов
      user1 = user2;
      user2.Name = "Natalya";
      user2.Age = 25;
      Console.Write("\nuser1: ");
      user1.WriteUserInfo();
      Console.Write("user2: ");
      user2.WriteUserInfo();
      Console.ReadLine();
    }
}
```

Результат выполнения:

A screenshot of a Windows console application window. The title bar shows the file path: file:///C:/projects/test/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1.... The console output displays four lines of text: 'user1: Имя: Alexandr, возраст: 26', 'user2: Имя: Elena, возраст: 22', 'user1: Имя: Elena, возраст: 22', and 'user2: Имя: Natalya, возраст: 25'. The text is white on a black background.

```
user1: Имя: Alexandr, возраст: 26
user2: Имя: Elena, возраст: 22

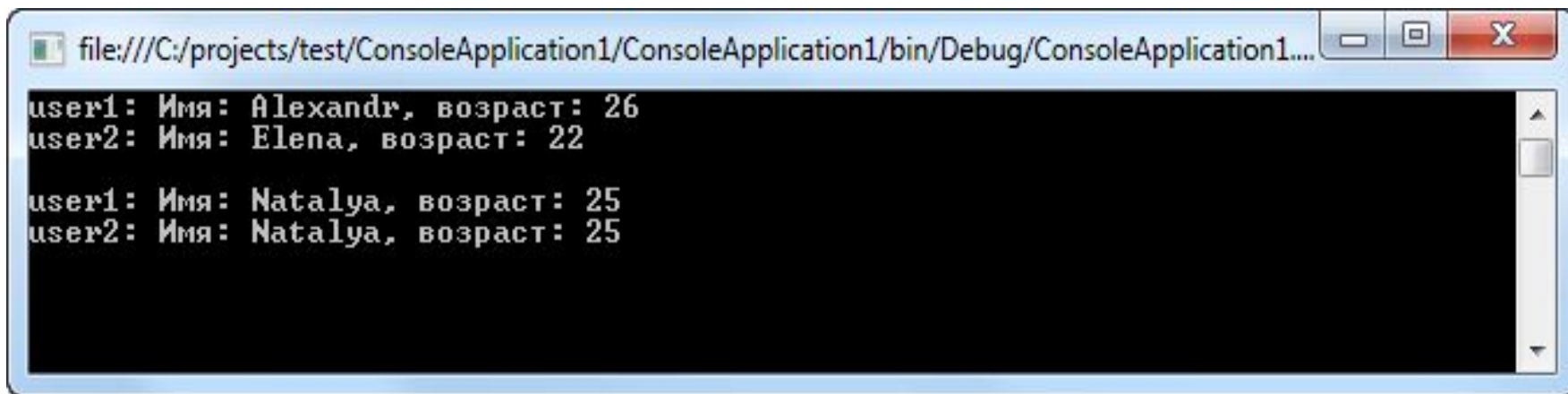
user1: Имя: Elena, возраст: 22
user2: Имя: Natalya, возраст: 25
```

Обратите внимание: когда одна структура присваивается другой, создается копия ее объекта.

В этом заключается одно из главных отличий структуры от класса. Когда ссылка на один класс присваивается ссылке на другой класс, в итоге ссылка в левой части оператора присваивания указывает на тот же самый объект, что и ссылка в правой его части.

А когда переменная одной структуры присваивается переменной другой структуры, создается копия объекта структуры из правой части оператора присваивания.

Поэтому, если бы в предыдущем примере использовался класс **UserInfo** вместо структуры, получился бы следующий результат:



```
file:///C:/projects/test/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1....
user1: Имя: Alexandr, возраст: 26
user2: Имя: Elena, возраст: 22

user1: Имя: Natalya, возраст: 25
user2: Имя: Natalya, возраст: 25
```

В листинге 2 приведен пример описания структуры, представляющей комплексное число.

Для экономии места из всех операций приведено только описание сложения.

Обратите внимание на перегруженный метод **ToString**: он позволяет выводить экземпляры структуры на консоль, поскольку неявно вызывается в методе **Console.WriteLine**.

Листинг 2 – Пример 2 структуры

```
using System;
namespace ConsoleApplication1
{
    struct Complex
    {
        public double re, im;
        public Complex(double re_, double im_)
        {re = re_; im = im_;} // МОЖНО ИСПОЛЬЗОВАТЬ this.re, this.im

        public static Complex operator + (Complex a, Complex b)
        {return new Complex(a.re + b.re, a.im + b.im);}

        public override string ToString()
        {
            return (string.Format("({0.2:0.##}:{1.2:0.##})", re, im));
        }
    }
}
```

```

class Class1
{ static void Main()
{
    Complex a = new Complex(1.2345, 5.6);
    Console.WriteLine("a = " + a);
    Complex b;
    b.re = 10; b.im = 1;
    Console.WriteLine("b = " + b);
    Complex c = new Complex();
    Console.WriteLine("c = " + c);
    c = a + b;
    Console.WriteLine("c = " + c);
}
}
}

```

Результат работы программы:



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The output of the program is displayed as follows:

```

a = <1,23;5,6>
b = <10;1>
c = <0;0>
c = <11,23;6,6>
Для продолжения нажмите любую клавишу . . . _

```

При выводе экземпляра структуры на консоль выполняется **упаковка**, то есть **неявное преобразование в ссылочный тип**. Упаковка применяется и в других случаях, когда структурный тип используется там, где ожидается ссылочный, например, при преобразовании экземпляра структуры к типу реализуемого ею интерфейса.

При обратном преобразовании – **из ссылочного типа в структурный** – выполняется **распаковка**.

При **присваивании структур** создается копия значений полей.

То же самое происходит и при передаче структур в качестве параметров по значению.

Для экономии ресурсов ничто не мешает передавать структуры в методы по ссылке с помощью ключевых слов **ref** или **out**.

Особенно значительный выигрыш в эффективности можно получить, используя **массивы структур** вместо **массивов классов**:

например, для массива из 100 экземпляров класса создается 101 объект,

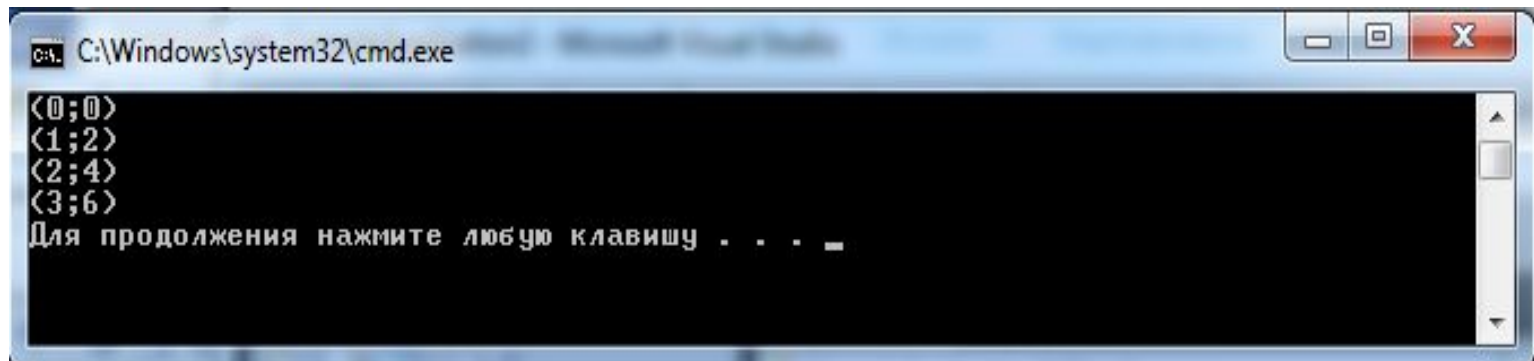
а для массива структур – один объект.

Приведем пример работы с массивом структур, описанных в предыдущем листинге:

```
Complex [] mas = new Complex[4];  
for (int i = 0; i < 4; ++i)  
{  
    mas[i].re = i;  
    mas[i].im = 2 * i;  
}
```

```
foreach (Complex elem in mas) Console.WriteLine(elem);
```

Если поместить этот фрагмент вместо тела метода **Main** в предыдущем листинге (пример структуры) получим следующий результат:



```
C:\Windows\system32\cmd.exe  
<0;0>  
<1;2>  
<2;4>  
<3;6>  
Для продолжения нажмите любую клавишу . . .
```

2. Перечисления

При написании программ часто возникает потребность определить несколько связанных между собой именованных констант, при этом их конкретные значения могут быть не важны.

Для этого удобно воспользоваться *перечисляемым типом данных*, все возможные значения которого задаются списком целочисленных констант, например:

```
enum Menu {Read, Write, Append, Exit}  
enum Радуга {Красный, Оранжевый, Желтый, Зеленый,  
Синий, Фиолетовый}
```

Для каждой константы задается ее символическое имя. По умолчанию константам присваиваются последовательные значения типа **int**, начиная с 0, но можно задать и собственные значения, например:

```
enum Nums {two = 2, three, four, ten = 10,  
           eleven, fifty = ten + 40};
```

Константам **three** и **four** присваиваются значения 3 и 4, константе **eleven** – 11.

Имена перечисляемых констант внутри каждого перечисления должны быть уникальными, а значения могут совпадать.

Преимущество перечисления перед описанием именованных констант состоит в том, что связанные константы нагляднее; кроме того, компилятор выполняет проверку типов, а интегрированная среда разработки подсказывает возможные значения констант, выводя их список.

Синтаксис перечисления:

```
[атрибуты][спецификаторы] enum имя_перечисления  
[ : базовый_тип]  
тело_перечисления [ ; ]
```

Спецификаторы перечисления имеют такой же смысл, как и для класса, причем допускаются только спецификаторы `new`, `public`, `protected`, `internal` и `private`.

Базовый тип – это тип элементов, из которых построено перечисление. По умолчанию используется тип `int`, но можно задать тип и явным образом, выбрав его среди целочисленных типов (кроме `char`), а именно: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` и `ulong` (когда значения констант невозможно или неудобно представлять с помощью типа `int`).

Тело перечисления состоит из *имен констант*, каждой из которых может быть присвоено значение. Если значение не указано, оно вычисляется прибавлением единицы к значению предыдущей константы. Константы по умолчанию имеют спецификатор доступа `public`.

Итак,

Перечисление (enumeration) – это определяемый пользователем целочисленный тип.

Объявляя перечисление, вы:

- специфицируете набор допустимых значений, которые могут принимать экземпляры перечислений;

- этим значениям присваиваете имена, понятные для пользователей.

Если где-то в коде попытаться присвоить экземпляру перечисления значение, не входящее в список допустимых, компилятор выдаст ошибку.

Существует, по крайней мере, **три преимущества** от применения перечислений вместо простых целых чисел:

- как упоминалось, перечисления облегчают сопровождение кода, гарантируя, что переменным будут присваиваться только легитимные, ожидаемые значения.

- перечисления делают код яснее, позволяя обращаться к целым значениям, называя их осмысленными именами вместо малопонятных "магических" чисел.

- перечисления облегчают ввод исходного кода. Когда вы собираетесь присвоить значение экземпляру перечислимого типа, то интегрированная среда Visual Studio с помощью средства IntelliSense отображает всплывающий список с допустимыми значениями, что позволяет сэкономить несколько нажатий клавиш и напомнить о возможном выборе значений.

При программировании перечислений нужно учесть:

1. Каждая символически обозначаемая константа в перечислении имеет целое значение.

Но, т.к. неявные преобразования перечислимого типа во встроенные целочисленные типы и обратно в C# не определены, то требуется **явное приведение типов**.

2. Поскольку перечисления обозначают целые значения, то их можно, например, использовать для управления оператором выбора **switch** или же оператором цикла **for**.

3. В перечислении для каждой последующей символически обозначаемой константы задается **целое значение, которое на единицу больше, чем у предыдущей константы**.

4. По умолчанию значение первой символически обозначаемой константы в перечислении равно нулю.

Рассмотрим пример использования перечислений (листинг 3).

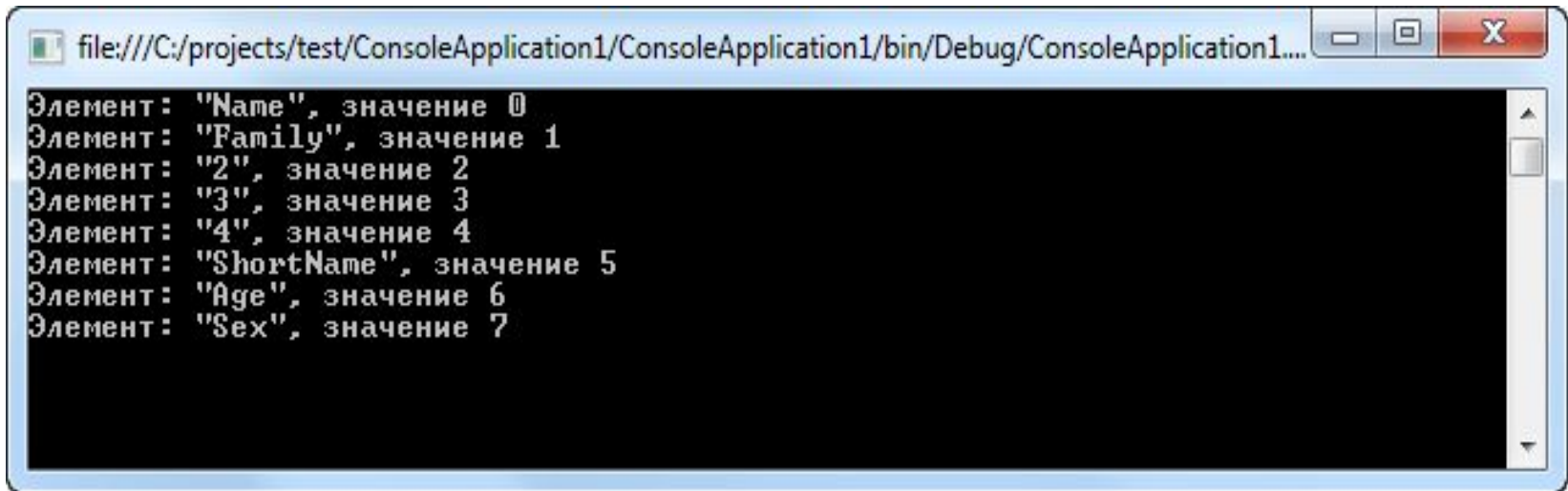
Листинг 3 – Пример перечисления

```
using System;
namespace ConsoleApplication1
{
    // Создать перечисление
    enum UI : long {Name, Family, ShortName = 5, Age, Sex}

    class Program
    {
        static void Main()
        {
            UI user1;
            for (user1 = UI.Name; user1 <= UI.Sex; user1++)
                Console.WriteLine("Элемент: \"{0}\", значение {1}",
user1,(int)user1);

            Console.ReadLine();
        }
    }
}
```

Результат выполнения:



```
file:///C:/projects/test/ConsoleApplication1/ConsoleApplication1/bin/Debug/ConsoleApplication1....  
Элемент: "Name", значение 0  
Элемент: "Family", значение 1  
Элемент: "2", значение 2  
Элемент: "3", значение 3  
Элемент: "4", значение 4  
Элемент: "ShortName", значение 5  
Элемент: "Age", значение 6  
Элемент: "Sex", значение 7
```

Значение одной или нескольких символически обозначаемых констант в перечислении можно задать с помощью инициализатора. Для этого достаточно указать после символического обозначения отдельной константы знак равенства и целое значение. Каждой последующей константе присваивается значение, которое на единицу больше значения предыдущей инициализированной константы. В приведенном выше примере инициализируется константа **ShortName**.

Перечисления часто используются как вложенные типы, идентифицируя значения из какого-либо ограниченного набора.

Листинг 4 – Пример перечисления

```
using System;
namespace ConsoleApplication1
{
    struct Боец
    { public enum Воинское_Звание
      {Рядовой, Сержант, Майор, Генерал}
      public string Фамилия;
      public Воинское_Звание Звание;
    }
    class Class1
    {static void Main()
      {
        Боец x;
        x.Фамилия = "Иванов";
        x.Звание = Боец.Воинское_Звание.Сержант;
        Console.WriteLine(x.Звание + " " + x.Фамилия);
      }
    }
}
```

Результат работы программы:

**Сержант
Иванов**

3. Операции с перечислениями

С переменными перечисляемого типа можно выполнять:

- арифметические операции (+, -, ++, --),
- логические поразрядные операции (^, &, |, ~),
- Сравнение с помощью операций отношения (<, <=, >, >=, ==, !=)
- получение размера в байтах (sizeof).

При использовании переменных перечисляемого типа в целочисленных выражениях и операциях присваивания требуется *явное преобразование типа*.

Переменной перечисляемого типа можно присвоить любое значение, представимое с помощью базового типа, то есть не только одно из значений, входящих в тело перечисления.

Присваиваемое значение становится новым элементом перечисления.

Перечисления удобно использовать для представления битовых флагов, например:

```
enum Flags : byte
{b0, b1, b2, b3 = 0x04, b4 = 0x08, b5 = 0x10, b6 = 0x20, b7
= 0x40}
Flags a = Flags.b2 | Flags.b4;
Console.WriteLine("a = {0} {1:X}", a, a);
++a; //{1:X} – шестнадцатеричный формат вывода
Console.WriteLine("a = {0} {1:X}", a, a);
int x = (int) a;
Console.WriteLine("x = {0} {1:X}", x, x);
Flags b = (Flags) 65;
Console.WriteLine("b = {0} {1:X}" b, b);
```

Результат работы этого фрагмента программы:



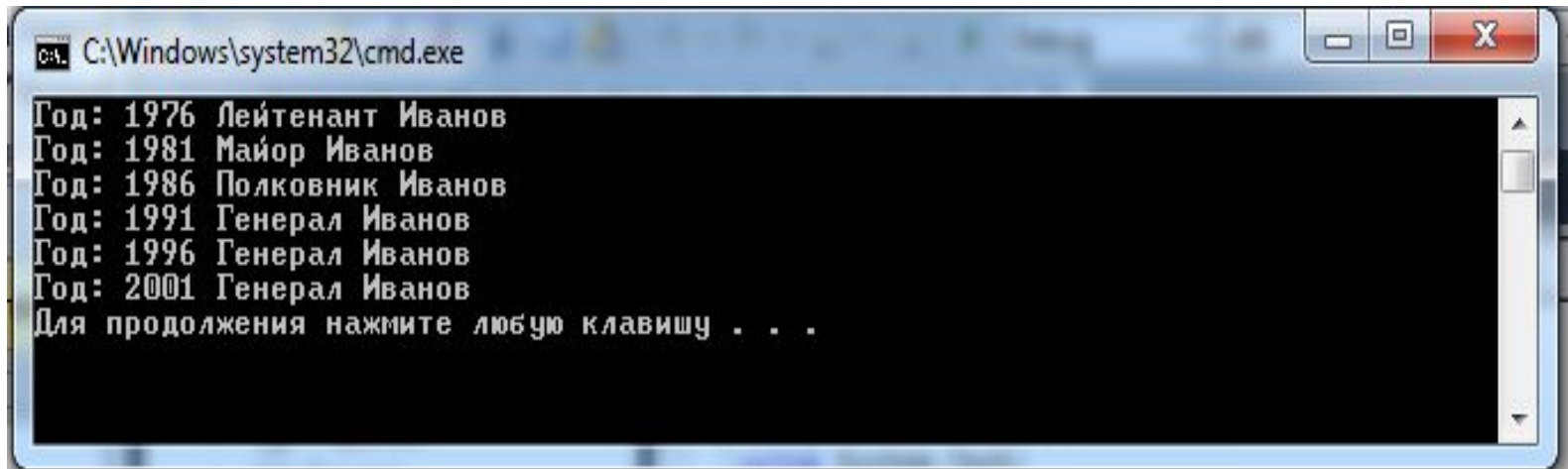
The screenshot shows a Windows command prompt window with the title bar "C:\Windows\system32\cmd.exe". The output of the program is displayed as follows:

```
a = 10 0A
a = 11 0B
x = 11 B
b = 65 41
Для продолжения нажмите любую клавишу . . .
```

Листинг 5– Операции с перечислениями

```
using System;
namespace ConsoleApplication1
{
    struct Боец
    {
        public enum Воинское_Звание
            {Рядовой, Сержант, Лейтенант, Майор, Полковник,
            Генерал}
        public string Фамилия;
        public Воинское_Звание Звание;}
    class Class1
    {
        static void Main()
        {
            Боец x;
            x.Фамилия = "Иванов";
            x.Звание = Боец.Воинское_Звание.Сержант;
            for (int i = 1976; i < 2006; i += 5)
            {
                if (x.Звание < Боец.Воинское_Звание.Генерал) ++x.Звание;
                Console.WriteLine("Год: {0} {1} {2}", i, x.Звание, x.
                Фамилия);
            }
        }
    }
}
```

Результат работы программы:



A screenshot of a Windows command prompt window. The title bar shows the path "C:\Windows\system32\cmd.exe". The window contains the following text:

```
Год: 1976 Лейтенант Иванов  
Год: 1981 Майор Иванов  
Год: 1986 Полковник Иванов  
Год: 1991 Генерал Иванов  
Год: 1996 Генерал Иванов  
Год: 2001 Генерал Иванов  
Для продолжения нажмите любую клавишу . . .
```

The text is displayed in a monospaced font on a black background. A vertical scrollbar is visible on the right side of the window.

4. Базовый класс `System.Enum`

Все перечисления в C# являются потомками базового класса `System.Enum`, который снабжает их некоторыми полезными методами.

1. Статический метод `GetName` позволяет получить символическое имя константы по ее номеру, например:

```
Console.WriteLine(Enum.GetName(typeof(Flags), 8));    // b4
Console.WriteLine(Enum.GetName(typeof(Боец.
Воинское_Звание), 1));    // Сержант
```

Примечание – Операция `typeof` возвращает тип своего аргумента

2. Статические методы `GetNames` и `GetValues` формируют, соответственно, массивы имен и значений констант, составляющих перечисление, например:

```
Array names = Enum.GetNames(typeof(Flags));
Console.WriteLine("Количество элементов в перечислении: " +
names.Length);
foreach (string elem in names) Console.Write("'" + elem);
Array values = Enum.GetValues(typeof(Flags));
foreach (Flags elem in values) Console.Write("'" + byte)elem);
```

3. Статический метод **IsDefined** возвращает значение **true**, если константа с заданным символическим именем описана в указанном перечислении, и **false** в противном случае, например:

```
if (Enum.IsDefined(typeof(Flags), "b5"))  
    Console.WriteLine("Константа с именем b5 существует");  
else    Console.WriteLine("Константа с именем b5 не  
существует");
```

4. Статический метод **GetUnderlyingType** возвращает имя базового типа, на котором построено перечисление.

Например, для перечисления **Flags** будет получено **System.Byte**:

```
Console.WriteLine(Enum.GetUnderlyingType(typeof(Flags)));
```

Выводы

Область применения **структур** – типы данных, имеющие небольшое количество полей, с которыми удобнее работать как со значениями, а не как со ссылками.

Накладные расходы на динамическое выделение памяти для экземпляров небольших классов могут весьма значительно снизить быстродействие программы, поэтому их эффективнее описывать как структуры.

Преимущество использования **перечислений** для описания связанных между собой значений состоит в том, что это более наглядно и инкапсулировано, чем россыпь именованных констант. Кроме того, компилятор выполняет проверку типов, а интегрированная среда разработки подсказывает возможные значения констант, выводя их список.