Класс char

В C# есть *символьный класс Char*, основанный на классе System.Char и использующий двухбайтную кодировку Unicode представления символов. Для этого типа в языке определены символьные константы – символьные литералы. Константу можно задавать:

- символом, заключенным в одинарные кавычки;
- escape-последовательностью, задающей код символа;
- Unicode-последовательностью, задающей Unicode-код символа.

Вот несколько примеров объявления символьных переменных и работы с ними:

```
public void TestChar()
{
       char ch1='A', ch2 ='\x5A', ch3='\u0058';
       char ch = new Char();
       int code; string s;
       ch = ch1;
       //преобразование символьного типа в тип int
       code = ch; ch1=(char) (code +1);
       //преобразование символьного типа в строку
       //s = ch;
       s = ch1.ToString()+ch2.ToString()+ch3.ToString();
       Console.WriteLine("s= {0}, ch= {1}, code = {2}",
              s, ch, code);
}//TestChar
```

Три символьные переменные инициализированы константами, значения которых заданы тремя разными способами. Переменная ch объявляется в объектном стиле, используя new и вызов конструктора класса. Тип char, как и все типы C#, является классом. Этот класс наследует свойства и методы класса Object и имеет большое число собственных методов.

Существуют ли преобразования между *классом char* и другими классами? Явные или неявные преобразования между *классами char* и string отсутствуют, но, благодаря методу ToString, переменные типа char стандартным образом преобразуются в тип string. Существуют *неявные преобразования типа* char в целочисленные типы, начиная с типа ushort. Обратные преобразования целочисленных типов в тип char также существуют, но они уже явные.

В результате работы процедуры TestChar строка s, полученная сцеплением трех символов, преобразованных в строки, имеет значение BZX, переменная ch равна A, а ее код – переменная code – 65.

Не раз отмечалось, что семантика присваивания справедлива при вызове методов и замене формальных аргументов на фактические. Приведу две процедуры, выполняющие взаимно-обратные операции – получение по коду символа и получение символа по его коду:

```
public int SayCode(char sym)
{
       return (sym);
}//SayCode

public char SaySym(object code)
{
       return ((char)((int)code));
}// SaySym
```

Как видите, в первой процедуре преобразование к целому типу выполняется неявно. Во второй - преобразование явное. Ради универсальности она слегка усложнена. Формальный

параметр имеет тип Object, что позволяет передавать ей в качестве аргумента код, заданный любым целочисленным типом. Платой за это является необходимость выполнять два явных преобразования.

Таблица 9.1. Статические методы и свойства класса Char	
Метод	**Описание**
`GetNumericValue`	Возвращает численное значение символа, если он является цифрой, и (`-1`) в противном случае
`GetUnicodeCategory`	Все символы разделены на категории. Метод возвращает Unicode категорию символа. Ниже приведен пример
`IsControl`	Возвращает `true`, если символ является управляющим
`IsDigit`	Возвращает `true`, если символ является десятичной цифрой
`IsLetter`	Возвращает `true`, если символ является буквой
`IsLetterOrDigit`	Возвращает `true`, если символ является буквой или цифрой
`IsLower`	Возвращает `true`, если символ задан в нижнем регистре
`IsNumber`	Возвращает `true`, если символ является числом (десятичной или шестнадцатиричной цифрой)
`IsPunctuation`	Возвращает `true`, если символ является знаком препинания
`IsSeparator`	Возвращает `true`, если символ является разделителем
`IsSurrogate`	Некоторые символы Unicode с кодом в интервале `[0x10000, 0x10FFF]` представляются двумя 16-битными "суррогатными" символами. Метод возвращает `true`, если символ является суррогатным
`IsUpper`	Возвращает `true`, если символ задан в верхнем регистре
`IsWhiteSpace`	Возвращает `true`, если символ является "белым пробелом". К белым пробелам, помимо пробела, относятся и другие символы, например, символ конца строки и символ перевода каретки
`Parse`	Преобразует строку в символ. Естественно, строка должна состоять из одного символа, иначе возникнет ошибка
`ToLower`	Приводит символ к нижнему регистру
`ToUpper`	Приводит символ к верхнему регистру
`MaxValue, MinValue`	Свойства, возвращающие символы с максимальным и минимальным кодом. Возвращаемые символы не имеют видимого образа

Класс char[] - массив символов

В языке C# определен *класс Char[]*, и его можно использовать для представления *строк постоянной длины*, как это делается в C++. Более того, поскольку массивы в C# динамические, то расширяется класс задач, в которых можно использовать массивы символов для представления строк. Так что имеет смысл разобраться, насколько хорошо C# поддерживает работу с таким представлением строк.

Прежде всего, ответим на вопрос, задает ли массив символов C# *строку C*, заканчивающуюся нулем? Ответ: нет, не задает. Массив char[] – это обычный массив. Более того,

его нельзя инициализировать строкой символов, как это разрешается в C++. Константа, задающая строку символов, принадлежит классу String, а в C# не определены взаимные преобразования между классами String и *Char[]*, даже явные. У класса String есть, правда, динамический метод ToCharArray, задающий подобное преобразование. Возможно также посимвольно передать содержимое переменной string в массив символов. Приведу пример:

```
public void TestCharArAndString()
{
	//массивы символов
	//char[] strM1 = "Hello, World!";
	//ошибка: нет преобразования класса string в класс char[]
	string hello = "Здравствуй, Мир!";
	char[] strM1 = hello.ToCharArray();
	PrintCharAr("strM1",strM1);
	//копирование подстроки
	char[] World = new char[3];
	Array.Copy(strM1,12,World,0,3);
	PrintCharAr("World",World);
	Console.WriteLine(CharArrayToString(World));
}//TestCharArAndString
```

Закомментированные операторы в начале этой процедуры показывают, что прямое присваивание строки массиву символов недопустимо. Однако метод ToCharArray, которым обладают строки, позволяет легко преодолеть эту трудность. Еще одну возможность преобразования строки в массив символов предоставляет статический метод Copy класса Array.

В нашем примере часть строки strM1 копируется в массив World. По ходу дела в методе вызывается процедура PrintCharAr класса Testing, печатающая массив символов как строку. Вот ее текст:

```
void PrintCharAr(string name,char[] ar)
{
	Console.WriteLine(name);
	for(int i=0; i < ar.Length; i++)
		Console.Write(ar[i]);
	Console.WriteLine();
}//PrintCharAr
```

Метод ToCharArray позволяет преобразовать строку в массив символов. К сожалению, обратная операция не определена, поскольку метод ToString, которым, конечно же, обладают все объекты *класса Char[]*, печатает информацию о классе, а не содержимое массива. Ситуацию легко исправить, написав подходящую процедуру. Вот текст этой процедуры CharArrayToString, вызываемой в нашем тестирующем примере:

```
string CharArrayToString(char[] ar)
{
	string result="";
	for(int i = 0; i< ar.Length; i++) result += ar[i];
	return(result);
}//CharArrayToString
```

Класс Char[], как и всякий класс-массив в C#, является наследником не только класса Object, но и класса Array, и, следовательно, обладает всеми методами родительских классов, подробно рассмотренных в предыдущей главе. А есть ли у него специфические методы, которые

позволяют выполнять операции над строками, представленными массивами символов? Таких специальных операций нет. Но некоторые перегруженные методы класса Array можно рассматривать как операции над строками. Например, метод Copy дает возможность выделять и заменять подстроку в теле строки. Методы IndexOf, LastIndexOf позволяют определить индексы первого и последнего вхождения в строку некоторого символа. К сожалению, их нельзя использовать для более интересной операции - нахождения индекса вхождения подстроки в строку. При необходимости такую процедуру можно написать самому. Вот как она выглядит:

```
int IndexOfStr( char[]s1, char[] s2)
{
	//возвращает индекс первого вхождения подстроки s2 в
	//строку s1
	int i =0, j=0, n=s1.Length-s2.Length; bool found = false;
	while( (i<=n) && !found)
	{
		j = Array.IndexOf(s1,s2[0],i);
		if (j <= n)
		{
			found=true; int k = 0;
			while ((k < s2.Length)&& found)
			{
				found =char.Equals(s1[k+j],s2[k]); k++;
			}
		}
		i=j+1;
	}
	if(found) return(j); else return(-1);
}//IndexOfStr
```

В реализации используется метод IndexOf класса Array, позволяющий найти начало совпадения строк, после чего проверяется совпадение остальных символов. Реализованный здесь алгоритм является самым очевидным, но далеко не самым эффективным.

А теперь рассмотрим процедуру, в которой определяются индексы вхождения символов и подстрок в строку:

```
public void TestIndexSym()
{
	char[] str1, str2;
	str1 = "рококо".ToCharArray();
	//определение вхождения символа
	int find, lind;
	find= Array.IndexOf(str1,'о');
	lind = Array.LastIndexOf(str1,'о');
	Console.WriteLine("Индексы вхождения о в рококо:{0},{1};
		", find, lind);
	//определение вхождения подстроки
	str2 = "рок".ToCharArray();
	find = IndexOfStr(str1,str2);
	Console.WriteLine("Индекс первого вхождения рок в
		рококо:{0}", find);
	str2 = "око".ToCharArray();
	find = IndexOfStr(str1,str2);
```

```
        Console.WriteLine("Индекс первого вхождения око в
              рококо:{0}", find);
}//TestIndexSym
```

В этой процедуре вначале используются стандартные методы класса Array для определения индексов вхождения символа в строку, а затем созданный метод IndexOfStr для определения индекса первого вхождения подстроки. Корректность работы метода проверяется на разных строках. Вот результаты ее работы.

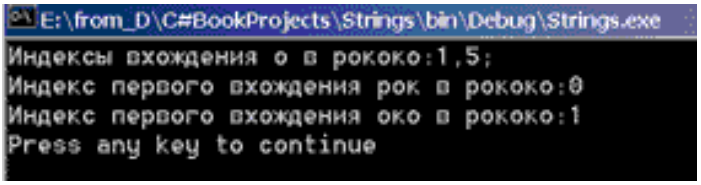

Рис. 9.3. Индексы вхождения подстроки в строку

С точки зрения регулярного программирования строковый тип данных `string` относится к числу самых важных в C#. Этот тип определяет и поддерживает символьные строки. В целом ряде других языков программирования строка представляет собой массив символов. А в `C#` строки являются объектами. Следовательно, тип `string` относится к числу ссылочных.