# Фундаментальные структуры данных

## Массивы

**Массив** – это структура данных с фиксированным и упорядоченным набором однотипных элементов. Доступ к элементу массива осуществляется по имени и номеру (индексу) этого элемента. Количество индексов определяет размерность массива.

Массивы бывают:

* одномерные (векторы),
* двумерные (матрицы).

Первые имеют один индекс, вторые – два.

А[i]

А – название массива.

[i] – индекс массива.

Если А – матрица, то она представлена в виде таблицы.

А[i, j], где

i – номер строки.

j – номер столбца.

Например, в языке С++ двумерный массив обозначается как A[i][j]

Обязательным критерием статического массива является однородность данных. Если данное условие не выполняется, массив является гетерогенным. Также существуют динамические массивы.

**Динамический массив** – это структура данных переменного размера, позволяющая в процессе выполнения программы автоматически добавлять и удалять элементы. Его максимальный размер может быть определен константой или определяться во время выполнения программы. В первом случае, динамический массив строится на базе статического массива, а во втором – переменного, то есть такого, размер которому назначается во время работы программы.

Для доступа к элементам индексного массива используются обычные целые числа, называемые индексами. У **ассоциативного массива** (словаря) эту функцию выполняют ключи. Они, в отличие от индексов, могут быть заданы не только числовым типом данных, но и, например строковым или булевым. Каждому элементу ассоциативного массива соответствует пара «ключ-значение» (key, value), и на нем определены четыре базовые операции:

* INSERT – операция добавления пары в массив;
* REASSIGN – операция изменения существующей пары;
* DELETE – операция удаления пары из массива;
* SEARCH – операция поиска пары в массиве.

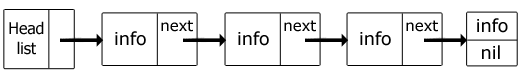
**Хеш-функция** – функция, преобразовывающая входную последовательность данных произвольного размера в выходную последовательность фиксированного размера. Процесс преобразования данных называется **хешированием**. Результат хеширования – хеш-код.

**Хеш-таблицей** называется структура данных, предназначенная для реализации ассоциативного массива, в котором адресация реализуется посредством хеш-функции.

## Списки

**Список** – абстрактный тип данных, реализующий упорядоченный набор значений. Списки отличаются от массивов тем, что доступ к их элементам осуществляется последовательно, в то время как массивы – структура данных произвольного доступа.

**Связный список** – структура данных, представляющая собой конечное множество упорядоченных элементов, связанных между собой посредством указателей. Каждый элемент структуры содержит поле с какой-либо информацией, а также указатель на следующий элемент.

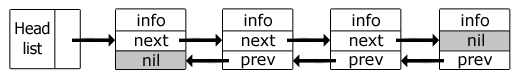


ОДНОСВЯЗНЫЙ СПИСОК

В последнем элементе вместо указателя содержится признак конца списка – nil.

Односвязный список не слишком удобен по причине возможности попадания лишь в следующую точку, двигаясь тем самым в конец.

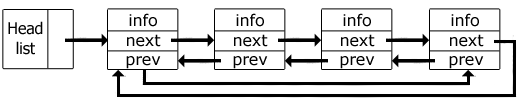
Когда кроме указателя на следующий элемент есть указатель на предыдущий, такой список называется **двусвязным**.



ДВУСВЯЗНЫЙ СПИСОК

Возможность двигаться вперед и назад полезна для выполнения некоторых операций, однако дополнительные указатели требуют задействования большего объема памяти.

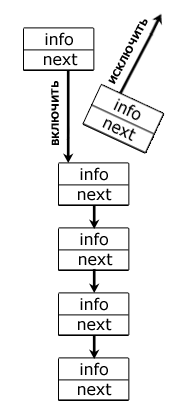
Кольцевой список – это подвид двух представленных видов списков.



КОЛЬЦЕВОЙ СПИСОК

## Стеки

Стек характерен тем, что получить доступ к его элементам можно лишь с одного конца, называемого вершиной стека. **Стек** – это структура данных, функционирующая по принципу LIFO (last in – first out, «пришел последним – вышел первым»).

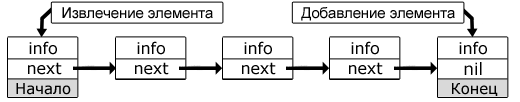


СТЕК

Для включения нужного элемента в пятую по счету ячейку необходимо исключить тот элемент, который занимает эту позицию.

## Очередь

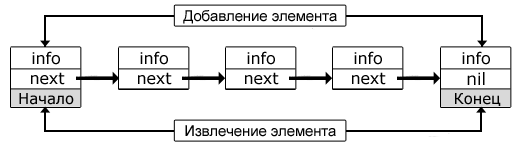
Структура данных «очередь» использует принцип организации FIFO (first in – first out, «пришел первым – ушел первым»). **Очередь** – это список, добавление элементов в который допустимо лишь в его конец, а их извлечение производится с другой стороны, называемой началом списка.



ОЧЕРЕДЬ

### Дек

Дек – это стек с двумя концами. Это означает, что данный вид списка позволяет добавлять элементы в начало и в конец, и то же самое справедливо для операции извлечения.

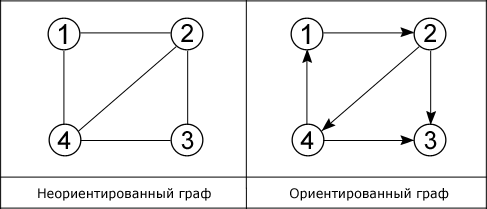


ДЕК

Данная структура работает одновременно по двум способам организации данных: LIFO и FIFO.

### Графы

**Граф** – совокупность точек, соединенных линиями. Точки – вершины (узлы), линии – ребра (дуги).



ГРАФЫ

Существуют следующие виды графов:

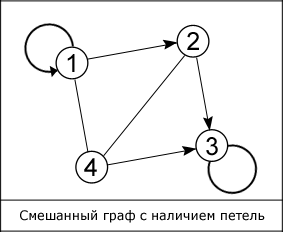
* ориентированные – ребра являются направленными, т. е. существует только одно доступное направление между двумя связными вершинами.
* неориентированные – ребра являются ненаправленными, т. е. из каждой вершины можно пройти в каждую и обратно.
* смешанные – характерны наличием как ориентированных, так и неориентированных ребер.

Степень входа вершины – количество входящих в нее ребер.

Степень выхода вершины – количество выходящих из нее ребер.

Ребра графа необязательно должны быть прямыми, а вершины обозначаться именно цифрами. К тому же встречаются такие графы, ребрам которых поставлено в соответствие конкретное значение. Они именуются взвешенными графами, а это значение – весом ребра.

Когда у ребра оба конца совпадают, т. е. ребро выходит из вершины F и входит в нее, то такое ребро называется петлей.



СМЕШАННЫЙ ГРАФ

G = (V, E)

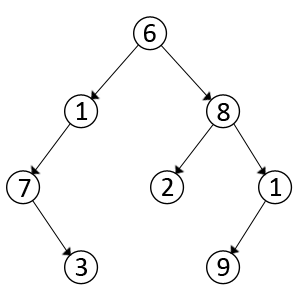
G – граф, V – вершины графа, Е – ребра графа.

ǀVǀ – порядок (число вершин).

ǀEǀ – размер графа (число ребер).

## Деревья

**Дерево** как математический объект – это абстракция из соименных единиц, встречающихся в природе.



ДЕРЕВО

2, 3, 9 – листья, т. е. узлы, не имеющие потомков (не ссылающиеся на какие-либо другие узлы).

6 – корневой узел в структуре дерева, т. е. единственная вершина, не имеющая предков (на нее не ссылается какая-либо другая вершина).

1, 1, 7, 8 – элементы, расположенные между корневым узлом и листьями. Они называются промежуточными узлами.

8 – поддерево, т. е. корневой узел, имеющий узлы-потомки.

# Алгоритмизация

# Сортировка

## Сортировка пузырьком

**Сортировка пузырьком (обменная сортировка)** – простой в реализации и малоэффективный алгоритм сортировки. Идея алгоритма заключается в том, что соседние элементы последовательности сравниваются между собой и меняются местами в случае необходимости.

## Сортировка выбором

**Сортировка выбором** – возможно, самый простой в реализации алгоритм сортировки. Пусть имеется массив А размером N, тогда сортировка выбором сводится к следующему:

* берем первый элемент последовательности A[i], здесь i – номер элемента, для первого i равен 1;
* находим минимальный (максимальный) элемент последовательности и запоминаем его номер в переменную key;
* если номер первого элемента и номер найденного элемента не совпадают, т. е. если key≠1, тогда два этих элемента обмениваются значениями, иначе никаких манипуляций не происходит;
* увеличиваем i на 1 и продолжаем сортировку оставшейся части массива, а именно с элемента с номером 2 по N, так как элемент A[1] уже занимает свою позицию.

## Сортировка вставками

**Сортировка вставками** – простой алгоритм сортировки, преимущественно использующийся в учебном программировании.

Рассмотрим алгоритм сортировки вставками на примере колоды игральных карт. Процесс их упорядочивания по возрастанию (в колоде карты расположены в случайном порядке) будет следующим. Обратим внимание на вторую карту, если ее значение меньше первой, то меняем эти карты местами, в противном случае карты сохраняют свои позиции, и алгоритм переходит к шагу 2. На 2-ом шаге смотрим на третью карту, здесь возможны четыре случая отношения значений карт:

* первая и вторая карта меньше третьей;
* первая и вторая карта больше третьей;
* первая карта уступает значением третьей, а вторая превосходит ее;
* первая карта превосходит значением третью карту, а вторая уступает ей.

В первом случае не происходит никаких перестановок. Во втором – вторая карта смещается на место третьей, первая на место второй, а третья карта занимает позицию первой. В предпоследнем случае первая карта остается на своем месте, в то время как вторая и третья меняются местами. Ну и наконец, последний случай требует рокировки лишь первой и третьей карт. Все последующие шаги полностью аналогичны расписанным выше.

## Сортировка Шелла

**«Сортировка Шелла»** – этот алгоритм можно рассматривать как обобщение пузырьковой сортировки, так и сортировки вставками.

Идея метода заключается в сравнении разделенных на группы элементов последовательности, находящихся друг от друга на некотором расстоянии. Изначально это расстояние равно d или N/2, где N — общее число элементов. На первом шаге каждая группа включает в себя два элемента расположенных друг от друга на расстоянии N/2; они сравниваются между собой, и, в случае необходимости, меняются местами. На последующих шагах также происходят проверка и обмен, но расстояние d сокращается на d/2, и количество групп, соответственно, уменьшается. Постепенно расстояние между элементами уменьшается, и на d=1 проход по массиву происходит в последний раз.

## Быстрая сортировка

Отличительной особенностью **быстрой сортировки** является операция разбиения массива на две части относительно опорного элемента. Например, если последовательность требуется упорядочить по возрастанию, то в левую часть будут помещены все элементы, значения которых меньше значения опорного элемента, а в правую элементы, чьи значения больше или равны опорному. Вне зависимости от того, какой элемент выбран в качестве опорного, массив будет отсортирован, но все же наиболее удачным считается ситуация, когда по обеим сторонам от опорного элемента оказывается примерно равное количество элементов. Если длина какой-то из получившихся в результате разбиения частей превышает один элемент, то для нее нужно рекурсивно выполнить упорядочивание, т. е. повторно запустить алгоритм на каждом из отрезков.

## Сортировка слиянием

В основе **сортировки слиянием** лежит принцип «разделяй и властвуй». Список разделяется на равные или практически равные части, каждая из которых сортируется отдельно. После чего уже упорядоченные части сливаются воедино.

## Гномья сортировка

**Гномья сортировка** основана на технике, используемой обыкновенным голландским садовым гномом. Вот как садовый гном сортирует ряд цветочных горшков. По существу, он смотрит на два соседних цветочных горшка, если они находятся в правильном порядке, то он переходит на один горшок вперед, иначе он меняет их местами и возвращается на один горшок назад. Граничные условия: если позади нет ни одного горшка – он шагает вперед, а если нет следующего горшка, тогда он закончил.

## Шейкерная сортировка (сортировка перемешиванием)

Перестановка элементов в **шейкерной сортировке** выполняется аналогично той же в пузырьковой сортировке, т. е. два соседних элемента, при необходимости, меняются местами.

# Поиск

## Линейный поиск

Слово «последовательный» содержит в себе основную идею метода. Начиная с первого, все элементы массива последовательно просматриваются и сравниваются с искомым. Если на каком-то шаге текущий элемент окажется равным искомому, тогда элемент считается найденным, и в качестве результата возвращается номер этого элемента, либо другая информация о нем.

## Двоичный поиск

Метод использует стратегию «разделяй и властвуй», а именно: заданная последовательность делится на две равные части и поиск осуществляется в одной из этих частей, которая потом также делится надвое, и так до тех пор, пока обнаружится наличие искомого элемента или его отсутствие. Использовать эту операцию, уменьшая каждый раз зону поиска вдвое, позволительно лишь исходя из того факта, что элементы последовательности заранее упорядочены.

## Интерполяционный поиск

В основе интерполяционного поиска лежит операция интерполирование. Интерполирование – нахождение промежуточных значений величины по имеющемуся дискретному набору известных значений. Интерполяционный поиск работает только с упорядоченными массивами; он похож на бинарный, в том смысле, что на каждом шаге вычисляется некоторая область поиска, которая, по мере выполнения алгоритма, сужается. Но в отличие от двоичного, интерполяционный поиск не делит последовательность на две равные части, а вычисляет приблизительное расположение ключа (искомого элемента), ориентируясь на расстояние между искомым и текущим значением элемента.

# Язык С#

## Краткий обзор элементов С#

**/\***

**Это простая программа на C#.**

**Назовем ее Exapmple.cs.**

**\*/**

Данные строки образуют комментарий. Содержимое комментария игнорируется компилятором. Данный комментарий называется многострочным. Всего в C# существует три стиля комментариев.

**using System;**

Эта строка означает, что в программе используется пространство имен System. Благодаря пространству имен одно множество имен отделяется от других.

**class Example {**

В этой строке ключевое слово **class** служит для объявления вновь определяемого класса. Класс является основной единицей инкапсуляции в С#, а **Example** – это имя класса. Определение класса начинается с открывающей фигурной скобки ( { ) и оканчивается закрывающей фигурной скобкой ( } ). Элементы, заключенные в эти фигурные скобки, являются членами класса.

**// Любая программа на С# начинается с вызова метода Main ().**

Данный комментарий является однострочным и представляет второй стиль комментариев, поддерживаемых в C#. Третий стиль комментариев, поддерживаемых в C#, применяется при создании документации.

**static void Main () {**

Эта строка начинается с метода **Main ()**. В C# подпрограмма называется методом. Именно с этой строки начинается выполнение программы.

Данная строка начинается с ключевого слова **static**. Метод, определяемый ключевым словом **static**, может вызываться до создания объекта его класса. Необходимость в этом объясняется тем, что метод **Main ()** вызывается при запуске программы. Ключевое слово **void** указывает на то, что метод **Main ()** не возвращает значение. Пустые круглые скобки после имени метода **Main** означают, что этому методу не передается никакой информации. Символ **{** обозначает начало тела метода **Main ()**.

**Console.WriteLine (“Простая программа на С#. “);**

Данная строка находится внутри метода **Main ()**. В этой строке осуществляется вывод на экран текстовой строки «Простая программа на C#.». Сам вывод выполняется встроенным методом **WriteLine ()**. В данном примере метод **WriteLine ()** выводит на экран строку, которая ему передается. Информация, передаваемая методу, называется аргументом. Помимо текстовых строк, метод **WriteLine ()** позволяет выводить на экран другие виды информации. Анализируемая строка начинается с **Console** – имени предопределенного класса, поддерживающего ввод-вывод на консоль. Сочетание обозначений **Console** и **WriteLine ()** указывает компилятору на то, что метод **WriteLine ()** является членом класса **Console**.

Как правило, операторы в C# заканчиваются точкой с запятой. Исключением из этого правила служат блоки, которые начинаются символом ( { ) и оканчиваются символом ( } ). Блоки обеспечивают механизм группирования операторов.

В C# различают прописные и строчные буквы. Поэтому если ввести вместо метода Main с большой буквы main с маленькой, получим ошибку компилятора о том, что не определена точка входа.

## Обработка синтаксических ошибок

В сообщении о синтаксической ошибке обычно указывается номер строки исходного текста программы, где была обнаружена ошибка, а также кратко описывается характер ошибки.

## Незначительное изменение программы

**using System;**

Формально данная строка не нужна в программе.

Например, вместо этого строку

**Console.WriteLine (“Простая программа на С#. “);**

можно переписать следующим образом.

**System.Console.WriteLine (“Простая программа на С#. “);**

Указывать пространство имен **System** всякий раз, когда используется член этого пространства, – довольно утомительное занятие, и поэтому большинство программистов вводят директиву **using System** в начале своих программ.

## Вторая простая программа

**Переменная** – это именованная область памяти, для которой может быть установлено значение. Она называется переменной потому, что ее значение может быть изменено по ходу выполнения программы.

В приведенной ниже программе создаются две переменные – **х** и **y**.

**// Эта программа демонстрирует применение переменных.**

**using System;**

**class Example2 {**

**static void Main () {**

**int x; // здесь объявляется переменная**

**int y; // здесь объявляется еще одна переменная**

**x = 100; // здесь переменной x присваивается значение 100**

**Console.WriteLine (“x содержит “ + x);**

**y = x / 2;**

**Console.Write (“y содержит x / 2: ”);**

**Console.WriteLine (y);**

**}**

**}**

Выполнение этой программы дает следующий результат.

**x содержит 100**

**y содержит x / 2: 50**

В этой программе вводится ряд новых понятий.

**int x; // здесь объявляется переменная**

Это переменная целочисленного типа с именем **х**. В C# все переменные должны объявляться до их применения. Кроме того, нужно обязательно указать тип значения, которое будет храниться в переменной. Для объявления в C# переменной целочисленного типа перед ее именем указывается ключевое слово **int**.

В целом, для объявления переменной служит следующий оператор:

**тип имя\_переменной;**

В C# оператор присваивания обозначается одиночным знаком равенства ( = ). Данный оператор выполняет копирование значения, расположенного справа от знака равенства, в переменную, находящуюся слева от него.

В следующей строке программы осуществляется вывод на экран текстовой строки “x содержит ” и значения переменной **х**.

**Console.WriteLine (“x содержит “ + x);**

В этом операторе знак + обозначает, что значение переменной **х** выводится вслед за предшествующей ему текстовой строкой. Если обобщить этот частный случай, то с помощью знака операции + можно организовать сцепление какого угодно числа элементов в одном операторе с вызовом метода **WriteLine ()**.

**y = x / 2;**

После выполнения данной строки в переменной **y** содержится значение 50. В C# поддерживаются все арифметические операции, в том числе

**+ сложение**

**- вычитание**

**\* умножение**

**/ деление**

Рассмотрим две оставшиеся строки программы.

**Console.Write (“y содержит x / 2: ”);**

**Console.WriteLine (y);**

В этих строках обнаруживаются еще две особенности. Во-первых, для вывода текстовой строки (“y содержит x / 2: ” на экран используется встроенный метод **Write ()**. После этой текстовой строки новая строка не следует. Это означает, что последующий вывод будет осуществлен в той же самой строке. Метод **Write ()** подобен методу **WriteLine ()**, за исключением того, что после каждого его вызова вывод не начинается с новой строки. И во-вторых, необходимо обратить внимание на то, что в вызове метода **WriteLine ()** указывается только переменная **y**.

Две или более переменных можно указать в одном операторе объявления. Нужно лишь разделить их запятой, например:

**int x, y; // обе переменные объявляются в одном операторе**

Чтобы отобразить результат программы необходимо в конце функции **Main ()** указать:

**Console.ReadLine ();**

## Другие типы данных

В переменных типа **int** могут храниться только целые числа. Их нельзя использовать в операциях с числами, имеющими дробную часть. Для операций с числами, имеющими дробную часть, в C# предусмотрены два типа данных с плавающей точкой: **float** и **double**. Они обозначают числовые значения с одинарной и двойной точностью соответственно.

Для объявления переменной типа **double** служит оператор **double result;**

где result – это имя переменной типа **double**. Поскольку переменная result имеет тип данных с плавающей точкой, в ней могут храниться такие числовые значения, как, например, 122,23, 0,034 или -19,0.

Для лучшего понимания отличий между типами данных **int** и **double** рассмотрим такой пример программы.

**/\* Эта программа демонстрирует отличия**

**между типами данных int и double.**

**\*/**

**using System;**

**class Example 3 {**

**static void Main () {**

**int ivar; // объявить целочисленную переменную**

**double dvar; // объявить переменную с плавающей точкой**

**ivar = 100; // присвоить переменной ivar значение 100**

**dvar = 100.0; // присовить переменной dvar значение 100.0**

**Console.WriteLine (“Исходное значение ivar: “ + ivar);**

**Console.WriteLine (“Исходное значение dvar: “ + dvar);**

**Console.WriteLine (); // вывести пустую строку**

**// разделить значения обеих переменных на 3.**

**ivar = ivar / 3;**

**dvar = dvar / 3.0;**

**Console.WriteLine (“Значение ivar после деления: ” + ivar);**

**Console.WriteLine (“Значение dvar после деления: ” + dvar);**

**}**

**}**

Ниже приведен результат выполнения приведенной выше программы.

**Исходное значение ivar: 100**

**Исходное значение dvar: 100.0**

**Значение ivar после деления: 33**

**Значение dvar после деления: 33.333333333333**

Чтобы вывести пустую строку, необходимо вызвать метод **Console.WriteLine ()** без аргументов.

Типы данных с плавающей точкой зачастую используются в операциях с реальными числовыми величинами, где обычно требуется дробная часть числа. Так, приведенная ниже программа вычисляет площадь круга, используя значение 3,1416 числа «пи».

**// Вычисление площади круга.**

**using System;**

**class Circle {**

**static void Main () {**

**double radius;**

**double area;**

**radius = 10.0;**

**area = radius \* radius \* 3.1416;**

**Console.WriteLine (“Площадь равна: ” + area);**

**}**

**}**

Выполнение этой программы дает следующий результат.

**Площадь равна: 314.16**

## Два управляющих оператора

Выполнение программы внутри метода происходит последовательно от одного оператора к другому, т.е. по цепочке сверху вниз. Этот порядок выполнения программы можно изменить с помощью различных управляющих операторов, поддерживаемых в C#.

### Условный оператор

С помощью условного оператора **if** в C# можно организовать выборочное выполнение части программы.

**if (условие) оператор;**

Здесь условие представляет собой булево, т.е. логическое, выражение, принимающее одно из двух значений: «истина» или «ложь». Если условие истинно, то оператор выполняется. Если условие ложно, то выполнение программы происходит, минуя оператор. Ниже приведен пример применения условного оператора.

**if (10 < 11) Console.WriteLine (“10 меньше 11”);**

Теперь рассмотрим другой пример.

**if (10 < 9) Console. WriteLine (“не подлежит выводу”);**

В данном примере 10 не меньше 9. Следовательно, вызов метода **WriteLine ()** не произойдет.

В C# определен полный набор операторов отношения, которые можно использовать в условных выражениях.

**< Меньше**

**<= Меньше или равно**

**> Больше**

**>= Больше или равно**

**== Равно**

**!= Не равно**

Далее следует пример еще одной программы, демонстрирующей применение условного оператора **if**.

**// Продемонстрировать применение условного оператора if.**

**using System;**

**class IfDemo {**

**static void Main () {**

**int a, b, c;**

**a = 2;**

**b = 3;**

**if (a < b) Console.WriteLine (“a меньше b”);**

**// Не подлежит выводу.**

**if (a == b) Console.WriteLine (“этого никто не увидит”);**

**Console.WriteLine ();**

**c = a – b; // с содержит -1**

**Console.WriteLine (“c содержит -1”);**

**if (c >= 0) Console.WriteLine (“значение с неотрицательно”);**

**if (c < 0) Console.WriteLine (“значение с отрицательно”);**

**Console.WriteLine ();**

**c = b – a; // с содержит 1**

**Console.WriteLine (“c содержит 1”);**

**if (c >= 0) Console.WriteLine (“значение с неотрицательно”);**

**if (c < 0) Console.WriteLine (“значение с отрицательно”);**

**}**

**}**

Вот к какому результату приводит выполнение данной программы.

**a меньше b**

**c содержит -1**

**значение с отрицательно**

**с содержит 1**

**значение с неотрицательно**

### Оператор цикла

Для повторного выполнения последовательности операций в программе можно организовать цикл. Рассмотрим оператор цикла **for**.

**for (инициализация; условие; итерация) оператор;**

Ниже приведен пример краткой программы, демонстрирующей применение данного оператора цикла.

**// Продемонстрировать применение оператора цикла for.**

**using System;**

**class ForDemo {**

**static void Main () {**

**int count;**

**for (count = 0; count < 5; count = count + 1)**

**Console.WriteLine (“Это подсчет: ” + count);**

**Console.WriteLine (“Готово!”);**

**}**

**}**

Вот как выглядит результат выполнения данной программы.

**Это подсчет: 0**

**Это подсчет: 1**

**Это подсчет: 2**

**Это подсчет: 3**

**Это подсчет: 4**

**Готово!**

В данном примере **count** выполняет роль переменной управления циклом. В инициализирующей части оператора цикла **for** задается нулевое значение этой переменной. В начале каждого шага цикла, включая и первый, проверяется условие **count < 5**. Если эта проверка дает истинный результат, то выполняется оператор, содержащий метод **WriteLine ()**. Далее выполняется итерационная часть оператора цикла **for**, где значение переменной **count** увеличивается на 1. Этот процесс повторяется до тех пор, пока значение переменной **count** не достигнет величины 5. В этот момент проверка упомянутого выше условия дает ложный результат, что приводит к завершению цикла.

В С# имеется специальный оператор инкремента, выполняющий приращение на 1 значения переменной, или так называемого операнда. Этот оператор обозначается как ++. Таким образом, оператор цикла **for** из приведенного выше примера программы обычно записывается в следующем виде.

**for (count = 0; count < 5; count ++)**

В С# имеется также оператор декремента, обозначаемый двумя дефисами --. Этот оператор уменьшает значение операнда на 1.

## Использование кодовых блоков

Еще одним важным элементом C# является кодовый блок, который представляет собой группу операторов. Для его организации достаточно расположить операторы между открывающей и закрывающей фигурными скобками. Как только кодовый блок будет создан, он станет логическим элементом, который можно использовать в любом месте программы, где применяется одиночный оператор. Так, кодовый блок может служить адресатом операторов **if** и **for**. Рассмотрим следующий оператор **if**.

**if (w < h) {**

**v = w \* h;**

**w = 0;**

**}**

Если в данном примере кода значение переменной **w** меньше значения переменной **h**, то оба оператора выполняются в кодовом блоке. Они образуют внутри кодового блока единый логический элемент, причем один не может выполняться без другого. Таким образом, если требуется логически связать два (или более) оператора, то для этой цели следует создать кодовый блок.

Ниже приведен пример программы, в которой кодовый блок служит для того, чтобы исключить деление на нуль.

**// Продемонстрировать применение кодового блока.**

**using System;**

**class BlockDemo {**

**static void Main () {**

**int i, j, d;**

**i = 5;**

**j = 10;**

**// Адресатом этого оператора if служит кодовый блок.**

**if (i != 0) {**

**Console.WriteLine (“i не равно нулю”);**

**d = j / i;**

**Console.WriteLine (“j/i равно: ” +d);**

**}**

**}**

**}**

Вот к какому результату приводит выполнение данной программы.

**i не равно нулю**

**j / i равно 2**

В данном примере адресатом оператора **if** служит кодовый блок, а не единственный оператор. Если условие, управляющее оператором **if**, оказывается истинным, то выполняются три оператора в кодовом блоке.

Рассмотрим пример, где кодовый блок служит для вычисления суммы и произведения чисел от 1 до 10.

**// Вычислить сумму и произведение чисел от 1 до 10**

**using System;**

**class ProdSum {**

**static void Main () {**

**int prod;**

**int sum;**

**int i;**

**sum = 0;**

**prod = 1;**

**for (i = 1; i <= 10; i++) {**

**sum = sum + i;**

**prod = prod \* i;**

**}**

**Console.WriteLine (“Сумма равна: ” +sum);**

**Console.WriteLine (“Произведение равно: ” +prod);**

**}**

**}**

Ниже приведен результат выполнения данной программы.

**Сумма равна: 55**

**Произведение равно: 3628800**

В данном примере внутри кодового блока организуется цикл для вычисления суммы и произведения. В отсутствие такого блока для достижения того же самого результата пришлось бы организовать два отдельных цикла.

Кодовые блоки не снижают эффективность программ во время их выполнения.

## Точка с запятой и оформление исходного текста программы

В C# точка с запятой обозначает конец оператора. Также конец строки не означает конец оператора. Например.

**x = y;**

**y = y + 1;**

**Console.WriteLine (x + “ “ + y);**

означают то же самое, что и строка кода

**x = y; y = y + 1; Console.WriteLine (x + “ “ + y);**

Более того, составные элементы оператора можно располагать в отдельных строках. Например, следующий фрагмент кода считается в С# вполне допустимым.

**Console.WriteLine (“Это длинная строка вывода” +**

**x + y + z +**

**“дополнительный вывод”);**

Такое разбиение длинных строк нередко применяется для того, чтобы сделать исходный текст программы более удобным для чтения.

## Ключевые слова С#

В С# определены два общих типа ключевых слов: зарезервированные и контекстные. Зарезервированные ключевые слова нельзя использовать в именах переменных, классов и методов. Иногда их еще называют зарезервированными словами или зарезервированными идентификаторами. Помимо этого, в C# определены контекстные ключевые слова, которые приобретают особое значение в определенном контексте. В таком контексте они выполняют роль ключевых слов, а вне его они могут использоваться в именах других элементов программы, например, в именах переменных. Использования их в любых других целях необходимо избегать.

Некоторые из зарезервированных ключевых слов: **abstract, byte, class, delegate, event, fixed, if, internal, new, override, readonly, short, struct, try, unsafe, void**.

Некоторые из контекстных ключевых слов: **add, group, var, partial, remove, where, select, join, set, get, global, value**.

## Идентификаторы

В C# идентификатор представляет собой имя, присваиваемое методу, переменной или любому другому определяемому пользователем элементу программы. Идентификаторы могут состоять из одного или нескольких символов. Имена переменных могут начинаться с любой буквы алфавита или знака подчеркивания. Далее может следовать буква, цифра или знак подчеркивания. С помощью знака подчеркивания можно повысить удобочитаемость имени переменной, как, например, line\_count. Но идентификаторы, содержащие два знака подчеркивания кряду, например, max\_\_value, зарезервированы для применения в компиляторе. Прописные и строчные буквы в C# различаются. Так, например, myvar и MyVar – это разные имена переменных.

Примеры допустимых идентификаторов: **Test, x, y2, MaxLoad, up, \_top, my\_var, sample23**.

Необходимо помнить, что идентификатор не может начинаться с цифры. Например, 12х – недействительный идентификатор.

Несмотря на то что зарезервированные ключевые слова нельзя использовать в качестве идентификаторов, в C# разрешается применять ключевое слово с предшествующим знаком @ в качестве допустимого идентификатора. Например, @for – действительный идентификатор. В этом случае в качестве идентификатора фактически служит ключевое слово for, а знак @ просто игнорируется. Ниже приведен пример программы, демонстрирующей применение идентификатора со знаком @.

**// Продемонстрировать применение идентификатора со знаком @.**

**using System;**

**class IDTest {**

**static void Main () {**

**int @if; // применение ключевого слова if в качестве идентификатора**

**for (@if = 0; @if < 10; @if++)**

**Console.WriteLine (“@if равно ” +@if);**

**}**

**}**

Приведенный ниже результат выполнения этой программы подтверждает, что **@if** правильно интерпретируется в качестве идентификатора.

**@if равно 0**

**@if равно 1**

**@if равно 2**

**@if равно 3**

**@if равно 4**

**@if равно 5**

**@if равно 6**

**@if равно 7**

**@if равно 8**

**@if равно 9**

Откровенно говоря, применять ключевые слова со знаком @ в качестве идентификаторов не рекомендуется, кроме особых случаев. Кроме того, знак @ может предшествовать любому идентификатору, но такая практика программирования считается плохой.

## Библиотека классов среды .NET Framework

## Типы значений в С#

В C# имеются две общие категории встроенных типов данных: типы значений и ссылочные типы. Они отличаются по содержимому переменной. Если переменная относится к типу значений, то она содержит само значение, например, 3,1416 или 212. Если же переменная относится к ссылочному типу, она содержит ссылку на значение. Наиболее распространенным примером использования ссылочного типа является класс.

Помимо простых типов, в C# определены еще три категории типов значений: перечисления, структуры и обнуляемые типы.

### Целочисленные типы

В C# определены девять целочисленных типов: **char, byte, sbyte, short, ushort, int, uint, long, ulong**. При этом тип **char** применяется, главным образом, для представления символов. Остальные восемь целочисленных типов предназначены для числовых расчетов.

В C# определены оба варианта различных целочисленных типов: со знаком и без знака. Целочисленные типы со знаком отличаются от аналогичных типов без знака способом интерпретации старшего разряда целого числа. Так, если в программе указано целочисленное значение со знаком, то компилятор C# сгенерирует код, в котором старший разряд целого числа используется в качестве флага знака. Число считается положительным, если флаг знака равен 0, и отрицательным, если он равен 1. Отрицательные числа практически всегда представляются методом дополнения до двух, в соответствии с которым все двоичные разряды отрицательного числа сначала инвертируются, а затем к этому числу добавляется 1.

Целочисленные типы со знаком имеют большое значение для очень многих алгоритмов, но по абсолютной величине они наполовину меньше своих аналогов без знака. Вот как, например, выглядит число 32767 типа **short** в двоичном представлении.

**0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1**

Если установить старший разряд этого числа равным 1, чтобы получить значение со знаком, то оно будет интерпретировано как -1, принимая во внимание формат дополнения до двух. Но если объявить его как значение типа **ushort**, то после установки в 1 старшего разряда оно станет равным 65535.

Вероятно, самым распространенным в программировании целочисленным типом является тип **int**. Переменные типа **int** нередко используются для управления циклами, индексирования массивов и математических расчетов общего назначения. Когда же требуется целочисленное значение с большим диапазоном представления чисел, чем у типа **int**, то для этой цели имеется целый ряд других целочисленных типов. Так, если значение нужно сохранить без знака, то для него можно выбрать тип **uint**, для больших значений со знаком – тип **long**, а для больших значений без знака – тип **ulong**. В качестве примера ниже приведена программа, вычисляющая расстояние от Земли до Солнца в дюймах. Для хранения столь большого значения в ней используется переменная **long**.

**// Вычислить расстояние от Земли до Солнца в дюймах.**

**using System;**

**class Inches {**

**static void Main () {**

**long inches;**

**long miles;**

**miles = 93000000; // 93000000 миль до Солнца**

**// 5280 футов в миле, 12 дюймов в футе.**

**inches = miles \* 5280 \* 12;**

**Console.WriteLine(“Расстояние до Солнца: ” + inches + “ дюймов.”);**

**}**

**}**

Вот так выглядит результат выполнения этой программы.

**Расстояние до Солнца: 5892480000000 дюймов**.

Самыми мелкими целочисленными типами являются **byte** и **sbyte**. Тип **byte** представляет целые значения без знака в пределах от 0 до 255. Переменные типа **byte** особенно удобны для обработки исходных двоичных данных, например, байтового потока, поступающего от некоторого устройства. Ниже приведен пример программы, в которой переменная типа **byte** используется для управления циклом, где суммируются от 1 до 100.

**// Использовать тип byte.**

**using System;**

**class Use\_byte {**

**static void Main () {**

**byte x;**

**int sum;**

**sum = 0;**

**for (x = 1; x <= 100; x++)**

**sum = sum + x;**

**Console.WriteLine(“Сумма чисел от 1 до 100 равна ” +sum);**

**}**

**}**

Результат выполнения этой программы выглядит следующим образом.

**Сумма чисел от 1 до 100 равна 5050**

В приведенном выше примере программы цикл выполняется только от 1 до 100, что не превышает диапазон представления чисел для типа **byte**, и поэтому для управления этим циклом не требуется переменная более крупного типа.

Если же требуется целое значение, большее чем значение **byte** или **sbyte**, но меньшее, чем значение типа **int** и **uint**, то для него можно выбрать тип **short** или **ushort**.

### Типы для представления чисел с плавающей точкой

Типы с плавающей точкой позволяют представлять числа с дробной частью. В C# имеются две разновидности типов данных с плавающей точкой: **float** и **double**. Они представляют числовые значения с одинарной и двойной точностью соответственно. Так, разрядность типа **float** составляет 32 бита, что приближенно соответствует диапазону представления чисел от 5Е-45 до 3,4E+38. Разрядность типа **double** составляет 64 бита, что приближенно соответствует диапазону представления чисел от 5Е-324 до 1,7Е+308.

В программировании на C# чаще применяется тип **double**, в частности, потому, что во многих математических функциях из библиотеки классов C#, которая одновременно является библиотекой классов для среды .NET Framework, используются числовые значения типа **double**. Например, метод Sqrt(), определенный в библиотеке классов System.Math, возвращает значение типа **double**, которое представляет собой квадратный корень из аргумента типа **double**, передаваемого данному методу. В приведенном ниже примере программы метод Sqrt() используется для вычисления радиуса окружности по площади круга.

**// Определить радиус окружности по площади круга.**

**using System;**

**class FindRadius {**

**static void Main () {**

**double r;**

**double area;**

**area = 10.0;**

**r = Math.Sqrt (area / 3.1416);**

**Console.WriteLine(“Радиус равен ” +r);**

**}**

**}**

Результат выполнения этой программы выглядит следующим образом.

**Радиус равен 1.78412203012729**

В приведенном выше примере программы следует обратить внимание на вызов метода Sqrt(). Как упоминалось выше, метод Sqrt() относится к классу Math, поэтому в его вызове имя Math предшествует имени самого метода. Аналогичным образом имя класса Console предшествует имени метода WriteLine() в его вызове. При вызове некоторых, хотя и не всех, стандартных методов обычно указывается имя их класса, как показано в следующем примере.

В следующем примере программы демонстрируется применение нескольких тригонометрических функций, которые относятся к классу Math и входят в стандартную библиотеку классов C#. Они также оперируют данными типа **double**. В этом примере на экран выводятся значения синуса, косинуса и тангенса угла, измеряемого в пределах от 0,1 до 1,0 радиана.

**// Продемонстрировать применение тригонометрических функций.**

**using System;**

**class Trigonometry {**

**static void Main () {**

**double theta; // угол в радианах**

**for (theta = 0.1; theta <= 1.0; theta = theta + 0.1) {**

**Console.WriteLine (“Синус угла ” + theta + “равен ” + Math.Sin (theta));**

**Console.WriteLine (“Коcинус угла ” + theta + “равен ” + Math.Cos (theta));**

**Console.WriteLine (“Тангенс угла ” + theta + “равен ” + Math.Tan (theta));**

**Console.WriteLine ();**

**}**

**}**

**}**

Ниже приведена лишь часть результата выполнения данной программы.

**Синус угла 0,1 равен 0,0998334166468282**

**Косинус угла 0,1 равен 0,995004165278026**

**Тангенс угла 0,1 равен 0,100334672085451**

**Синус угла 0,2 равен 0,198669330795061**

**Косинус угла 0,2 равен 0,980066577841242**

**Тангенс угла 0,2 равен 0,202710035508673**

Для вычисления синуса, косинуса и тангенса угла в приведенном выше примере были использованы стандартные методы **Math.Sin(), Math.Cos()** и **Math.Tan()**. Как и метод Math.Sqrt(), эти тригонометрические методы вызываются с аргументом типа **double** и возвращают результат того же типа. Вычисляемые углы должны быть указаны в радианах.

### Десятичный тип данных

Тип **decimal** применяется в финансовых расчетах. Он имеет разрядность 128 бит для представления числовых значений в пределах от 1E-28 до 7,9Е+28. Для обычных арифметических вычислений с плавающей точкой характерны ошибки округления десятичных значений. Эти ошибки исключаются при использовании типа **decimal**, который позволяет представить числа с точностью до 28 (а иногда и 29) десятичных разрядов. Благодаря тому что этот тип данных способен представлять десятичные значения без ошибок округления, он особенно удобен для расчетов, связанных с финансами.

Ниже приведен пример программы, в которой тип **decimal** используется в конкретном финансовом расчете. В этой программе цена со скидкой рассчитывается на основании исходной цены и скидки в процентах.

**// Использовать тип decimal для расчета скидки.**

**using System;**

**class UseDecimal {**

**static void Main () {**

**decimal price;**

**decimal discount;**

**decimal discounted\_price;**

**// Рассчитать цену со скидкой.**

**price = 19.95m;**

**discount = 0.15m; // норма скидки составляет 15%**

**discounted\_price = price – (price \* discount);**

**Console.WriteLine (“Цена со скидкой: $” +discounted\_price);**

**}**

**}**

Результат выполнения этой программы выглядит следующим образом.

**Цена со скидкой: $16.9575**

Обратите внимание на то, что значения констант типа **decimal** в приведенном выше примере программы указываются с суффиксом m. Дело в том, что без суффикса m эти значения интерпретировались бы как стандартные константы с плавающей точкой, которые несовместимы с типом данных **decimal**. Тем не менее, переменной типа **decimal** можно присвоить целое значение без суффикса m, например, 10.

Рассмотрим еще один пример применения типа **decimal**. В нем рассчитывается будущая стоимость капиталовложений с фиксированной нормой прибыли в течение ряда лет.

**/\* Применить тип decimal для расчета будущей стоимости**

**капиталовложений.**

**\*/**

**using System;**

**class FutVal {**

**static void Main () {**

**decimal amount;**

**decimal rate\_of\_return;**

**int years, i;**

**amount = 1000.0M;**

**rate\_of\_return = 0.07M;**

**years = 10;**

**Console.WriteLine (“Первоначальные капиталовложения: $” +amount);**

**Console.WriteLine (“Норма прибыли: ” +rate\_of\_return);**

**Console.WriteLine (“В течение ” +years+ “лет”);**

**for (i = 0; i < years; i++)**

**amount = amount + (amount \* rate\_of\_return);**

**Console.WriteLine (“Будущая стоимость равна $” +amount);**

**}**

**}**

Вот как выглядит результат выполнения этой программы.

**Первоначальные капиталовложения: $1000**

**Норма прибыли: 0.07**

**В течение 10 лет**

**Будущая стоимость равна $1967.151357289565322490000**

Обратите внимание на то, что результат выполнения приведенной выше программы представлен с точностью до целого ряда десятичных разрядов, т.е. с явным избытком по сравнению с тем, что обычно требуется.

### Символы

В C# символы представлены не 8-разрядным кодом, как во многих других языках программирования, а 16-разрядным кодом, который называется уникодом (Unicode). Например, в китайском языке употребляются довольно обширные наборы символов, которые нельзя представить 8-разрядным кодом. Для преодоления этого ограничения в C# определен тип **char**, представляющий 16-разрядные значения без знака в пределах от 0 до 65535. При этом стандартный набор символов в 8-разрядном коде ASCII является подмножеством уникода в пределах от 0 до 127. Следовательно, символы в коде ASCII по-прежнему остаются действительными в C#.

Для того чтобы присвоить значение символьной переменной, достаточно заключить это значение (т.е. символ) в одинарные кавычки. Так, в приведенном ниже фрагменте кода переменной ch присваивается символ X.

**char ch;**

**ch = ‘X’;**

Значение типа **char** можно вывести на экран с помощью метода **WriteLine()**. Например, в следующей строке кода на экран выводится значение переменной ch.

**Console.WriteLine(“Значение ch равно: ” +ch);**

Несмотря на то что тип **char** определен в C# как целочисленный, его не следует путать со всеми остальными целочисленными типами. Дело в том, что в C# отсутствует автоматическое преобразование символьных значений в целочисленные и обратно. Например, следующий фрагмент кода содержит ошибку.

**char ch;**

**ch = 88; // ошибка, не выйдет**

Ошибочность приведенного выше фрагмента кода объясняется тем, что 88 – это целое значение, которое не преобразуется автоматически в символьное. Для того чтобы операция присваивания целого значения символьной переменной оказалась допустимой, необходимо осуществить приведение типа, о котором речь пойдет далее.

### Логический тип данных

Тип **bool** представляет два логических значения: «ложь» и «истина». Эти логические значения обозначаются в C# зарезервированными словами false и true соответственно. Следовательно, переменная или выражение типа **bool** будет принимать одно из этих логических значений. Кроме того, в C# не определено взаимное преобразование логических и целых значений. Например, 1 не преобразуется в значение true, а 0 – в значение false.

В приведенном ниже примере программы демонстрируется применение типа **bool**.

**// Продемонстрировать применение типа bool.**

**using System;**

**class BoolDemo {**

**static void Main () {**

**bool b;**

**b = false;**

**Console.WriteLine (“b равно ” +b);**

**b = true;**

**Console.WriteLine (“b равно ” +b);**

**// Логическое значение может управлять оператором if.**

**if (b) Console.WriteLine (“Выполняется. ”);**

**b = false;**

**if (b) Console.WriteLine (“Не выполняется. ”);**

**// Результатом выполнения оператора отношения является логическое значение.**

**Console.WriteLine (“10 > 9” равно “ + (10 > 9));**

**}**

**}**

Эта программа дает следующий результат.

**b равно False**

**b равно True**

**Выполняется.**

**10 > 9 равно True**

В приведенной выше программе обнаруживаются три интересные особенности. Во-первых, при выводе логического значения типа **bool** с помощью метода **WriteLine()** на экране появляется значение true или false. Во-вторых, самого значения переменной типа **bool** достаточно для управления оператором **if**. Для этого не нужно, например, записывать оператор **if** следующим образом.

**if (b == true) …**

И в третьих, результатом выполнения оператора отношения является логическое значение. Именно поэтому в результате вычисления выражения 10 > 9 на экран выводится значение true. Кроме того, выражение 10 > 9 следует заключить в скобки, поскольку оператор + имеет более высокий приоритет, чем оператор >.

### Некоторые возможности вывода

В среде .NET Framework определен достаточно развитый механизм форматирования, позволяющий во всех деталях управлять выводом данных. Ознакомимся с некоторыми возможностями форматирования. Они позволяют указать, в каком именно виде следует выводить значения с помощью метода **WriteLine()**. Благодаря этому выводимый результат выглядит более привлекательно. Следует, однако, иметь ввиду, что механизм форматирования поддерживает намного больше возможностей, а не только те, которые рассматриваются в этом разделе.

При выводе список данных в предыдущих примерах программ каждый элемент списка приходилось отделять знаком +, как в следующей строке.

**Console.WriteLine (“Вы заказали ” + 2 +**

**“ предмета по цене $” + 3 + “ каждый.”);**

Конечно, такой способ вывода числовой информации удобен, но он не позволяет управлять внешним видом выводимой информации. Например, при выводе значения с плавающей точкой нельзя определить количество отображаемых десятичных разрядов. Рассмотрим оператор

**Console.WriteLine (“Деление 10/3 дает: ” + 10.0/3.0);**

который выводит следующий результат.

**Деление 10/3 дает: 3.33333333333333**

В одних случаях такого вывода может оказаться достаточно, а в других – он просто недопустим. Например, в финансовых расчетах после десятичной точки принято указывать лишь два десятичных разряда.

Для управления форматированием числовых данных служит другая форма метода **WriteLine()**, позволяющая встраивать информацию форматирования, как показано ниже.

**WriteLine (“форматирующая строка”, arg0, arg1, … , arg N);**

В этой форме аргументы метода **WriteLine()** разделяются запятой, а не знаком +. А **форматирующая строка** состоит из двух элементов: обычных печатаемых символов, предназначенных для вывода в исходном виде, а также спецификаторов формата. Последние указываются в следующей общей форме:

**{ argnum, width: fmt}**

где **argnum** – номер выводимого аргумента, начиная с нуля; **width** – минимальная ширина поля; **fmt** – формат. Параметры **width** и **fmt** являются необязательными.

Если во время выполнения в форматирующей строке встречается спецификатор формата, то вместо него подставляется и отображается соответствующий аргумент, обозначаемый параметром **argnum**. Таким образом, местоположение спецификатора формата в форматирующей строке определяет место отображения соответствующих данных. Параметры **width** и **fmt** указывать необязательно. Это означает, что в своей простейшей форме спецификатор формата обозначает конкретный отображаемый аргумент. Например, спецификатор {0} обозначает **arg0**, спецификатор {1} – аргумент **arg1** и т.д.

Начнем с самого простого примера. При выполнении оператора

**Console.WriteLine (“В феврале {0} или {1} дней.”, 28, 29);**

Получаем следующий результат.

**В феврале 28 или 29 дней**

Как видно, значение 28 подставляется вместо спецификатора {0}, а значение 29 – вместо спецификатора {1}. Следовательно, спецификаторы формата обозначают место в строке, где отображаются соответствующие аргументы. Кроме того, обратите внимание на то, что дополнительные значения разделяются запятой, а не знаком +.

Ниже приведен видоизмененный вариант предыдущего оператора, в котором указывается ширина полей.

**Console.WriteLine (“В феврале {0,10} или {1,5} дней.”, 28, 29);**

Выполнение этого оператора дает следующий результат.

**В феврале 28 или 29 дней**

Как видно, неиспользуемые части полей заполнены пробелами. Напомним, что минимальная ширина поля определяется параметром width. Если требуется, она может быть превышена при выводе результата.

Разумеется, аргументы, связанные с командой форматирования, не обязательно должны быть константами. Ниже приведен пример программы, которая выводит таблицу результатов возведения чисел в квадрат и куб. В ней команды форматирования используются для вывода соответствующих значений.

**// Применить команды форматирования.**

**using System;**

**class DisplayOptions {**

**static void Main () {**

**int i;**

**Console.WriteLine (“Число\tКвадрат\tКуб”);**

**for (i = 1; i < 10; i++)**

**Cosnole.WriteLine (“{0}\t{1}\t{2}”, i, i \* i, i \* i \* i);**

**}**

**}**

Результат выполнения этой программы выглядит следующим образом.

**Число Квадрат Куб**

**1 1 1**

**2 4 8**

**3 9 27**

**4 16 64**

**5 25 125**

**6 36 216**

**7 49 343**

**8 64 512**

**9 81 729**

В приведенных выше примерах сами выводимые значения не форматировались. Но ведь основное назначение спецификаторов формата – управлять внешним видом выводимых данных. Чаще всего форматированию подлежат следующие типы данных: с плавающей точкой и десятичный. Самый простой способу указать формат данных – описать шаблон, который будет использоваться в методе WriteLine(). Для этого указывается образец требуемого формата с помощью символов #, обозначающих разряды чисел. Кроме того, можно указать десятичную точку и запятые, разделяющие цифры. Ниже приведен пример более подходящего вывода результата деления 10 на 3.

**Console.WriteLine (“Деление 10/3 дает: {0:#.##}”, 10.0/3.0);**

Выполнение этого оператора приводит к следующему результату.

**Деление 10/3 дает: 3.33**

В данном примере шаблон #.## указывает методу **WriteLine()** отобразить два десятичных разряда в дробной части числа. Следует, однако, иметь ввиду, что метод **WriteLine()** может отобразить столько цифр слева от десятичной точки, сколько потребуется для правильной интерпретации выводимого значения.

Рассмотрим еще один пример. Оператор

**Console.WriteLine (“ {0:###, ###.##}, 123456.56);**

Дает следующий результат.

**123,456.56**

Для вывода денежных сумм, например, рекомендуется использовать спецификатор формата С.

**decimal balance;**

**balance = 12323.09m;**

**Console.WriteLine (“Текущий баланс равен {0:C}”, balance);**

Результат выполнения этого фрагмента кода выводится в формате денежных сумм, указываемых в долларах США.

**Текущий баланс равен $12,323.09**

Форматом С можно также воспользоваться, чтобы представить в более подходящем виде результат выполнения рассматривавшейся ранее программы расчета цены со скидкой.

**// Использовать спецификатор формата С для вывода результата в местной валюте.**

**using System;**

**class UseDecimal {**

**static void Main () {**

**decimal price;**

**decimal discount;**

**decimal discounted\_price;**

**// рассчитать цену со скидкой.**

**price = 19.95m;**

**discount = 0.15m; // норма скидки составляет 15%**

**discounted\_price = price – (price \* discount);**

**Console.WriteLine (“Цена со скидкой: {0:C} грн“, discounted\_price);**

**}**

**}**

Вот как теперь выглядит результат выполнения этой программы.

**Цена со скидкой: 16,96 грн**

### Литералы

В C# литералами называются постоянные значения, представленные в удобной для восприятия форме. Например, число 100 является литералом. Сами литералы и их назначение настолько понятны, что они применялись во всех предыдущих примерах программ без всяких пояснений.

В C# литералы могут быть любого простого типа. Представление каждого литерала зависит от конкретного типа. Как говорилось ранее, символьные литералы заключаются в одинарные кавычки. Например, ‘a’ и ‘%’ являются символьными литералами.

Целочисленные литералы указываются в виде чисел без дробной части. Например, 10 или -100 – это целочисленные литералы. Для обозначения литералов с плавающей точкой требуется указывать десятичную точку и дробную часть числа. Например, 11.123 – это литерал с плавающей точкой. Для вещественных чисел с плавающей точкой в C# допускается также использовать экспоненциальное представление.

У литералов должен быть также конкретный тип, поскольку C# является строго типизированным языком.

У целочисленных литералов должен быть самый мелкий целочисленный тип, которым они могут быть представлены, начиная с типа **int**. Таким образом, у целочисленных литералов может быть один из следующих типов: **int, uint, long** или **ulong** в зависимости от значения литерала. Литералы с плавающей точкой относятся к типу **double**.

Если вас не устраивает используемый по умолчанию тип литерала, вы можете явно указать другой его тип с помощью суффикса. Так, для указания типа **long** к литералу присоединяется суффикс 1 или L. К примеру, 12 – это литерал типа **int**, а 12L – литерал типа **long**. Для указания целочисленного типа без знака к литералу присоединяется суффикс u или U. Следовательно, 100 – это литерал типа **int**, а 100U – литерал типа **uint**. А для указания длинного целочисленного типа без знака к литералу присоединяется суффикс ul или UL. Например, 984375UL – это литерал типа **ulong**.

Кроме того, для указания типа **float** к литералу присоединяется суффикс F или f. Например, 10.19F – это литерал типа **float**. Можно даже указать тип **double**, присоединив к литералу суффикс d или D, хотя это излишне.

И наконец, для указания типа **decimal** к литералу присоединяется суффикс m или M. Например, 9.95M – это десятичный литерал типа **decimal**.

Несмотря на то что целочисленные литералы образуют по умолчанию значения типа **int, uint, long** или **ulong**, их можно присваивать переменным типа **byte**, **sbyte, short** или **ushort**, при условии, что присваиваемое значение может быть представлено целевым типом.

### Шестнадцатеричные литералы

В программировании иногда оказывается проще пользоваться системой счисления по основанию 16, чем по основанию 10. Такая система называется шестнадцатеричной. В ней используются числа от 0 до 9, а также буквы от A до F, которыми обозначаются десятичные числа 10, 11, 12, 13, 14 и 15. Например, десятичному числу 16 соответствует шестнадцатеричное число 10. Вследствие того что шестнадцатеричные числа применяются в программировании довольно часто, в C# разрешается указывать целочисленные литералы в шестнадцатеричном формате. Шестнадцатеричные литералы должны начинаться с символов 0x. Ниже приведены некоторые примеры шестнадцатеричных литералов.

**count = 0xFF; // 255 в десятичной системе**

**incr = 0x1a; // 26 в десятичной системе**

Например, 956(10) перевести в шестнадцатеричную систему.

956/16 = 59; 59/16 = **3**

59-16 = 43; 43-16 = 27; 27-16 = 11 (**В**)

59\*16 = 944; 956-944 = 12 (**С**)

Итог: 3ВС.

### Управляющие последовательности символов

Ряд символов, в том числе одинарные и двойные кавычки, имеют специальное назначение в C#, поэтому их нельзя использовать непосредственно. По этим причинам в C# предусмотрены специальные управляющие последовательности символов, иногда еще называемые константами с обратной косой чертой. Такие последовательности применяются вместо тех символов, которых они представляют.

**\a – звуковой сигнал (звонок)**

**\b – возврат на одну позицию**

**\f – перевод страницы (переход на новую страницу)**

**\n – новая строка (перевод строки)**

**\r – возврат каретки**

**\t – горизонтальная табуляция**

**\v – вертикальная табуляция**

**\0 – пустой символ**

**\’ – одинарная кавычка**

**\” – двойная кавычка**

**\\ – обратная косая черта**

Например, в следующей строке кода переменной ch присваивается символ табуляции.

**ch = ‘\t’;**

### Строковые литералы

В C# поддерживается еще один тип литералов – строковый. Строковый литерал представляет собой набор символов, заключенных в двойные кавычки. Например, следующий фрагмент кода:

**“это тест”**

представляет собой текстовую строку. Образцы подобных строк не раз встречались в приведенных выше примерах программ.

Можно также указать буквальный строковый литерал. Такой литерал начинается с символа **@**, после которого следует строка в кавычках. Содержимое строки в кавычках воспринимается без изменений и может быть расширено до двух и более строк. Это означает, что в буквальный строковый литерал можно включить символы новой строки, табуляции и прочие, не прибегая к управляющим последовательностям. Единственное исключение составляют двойные кавычки (“), для указания которых необходимо использовать две двойные кавычки подряд (“”). В приведенном ниже примере программы демонстрируется применение буквальных строковых литералов.

**// Продемонстрировать применение буквальных строковых литералов.**

**using System;**

**class Verbatim {**

**static void Main () {**

**Console.WriteLine (@ ”Это буквальный**

**строковый литерал,**

**занимающий несколько строк.”);**

**Console.WriteLine (@”А это вывод с табуляцией:**

**1 2 3 4**

**5 6 7 8**

**”);**

**Console.WriteLine (@”Отзыв программиста: “”Мне нравится C#.“””);**

**}**

Результат выполнения этой программы приведен ниже.

**Это буквальный**

**строковый литерал,**

**занимающий несколько строк.**

**А это вывод с табуляцией:**

**1 2 3 4**

**5 6 7 8**

**Отзыв программиста: “Мне нравится C#.”**

Преимущество буквальных строковых литералов заключается в том, что они позволяют указать в программе выводимый результат именно так, как он должен выглядеть на экране.

## Более подробное рассмотрение переменных

Переменные объявляются с помощью оператора следующей формы:

*тип имя\_переменной;*

где *тип* – тип данных, хранящихся в переменной; а *имя\_переменной* – это ее имя. Важно подчеркнуть, что возможности переменной определяются ее типом. Например, переменную типа **bool** нельзя использовать для хранения числовых значений с плавающей точкой. Кроме того, тип переменной нельзя изменять в течение срока ее существования. Так, переменную типа **int** нельзя преобразовать в переменную типа **char**.

Все переменные в C# должны быть объявлены до их применения. Это нужно для того, чтобы уведомить компилятор о типе данных, хранящихся в переменной, прежде чем он попытается правильно скомпилировать любой оператор, в котором используется переменная. Это позволяет также осуществлять строгий контроль типов в C#.

В C# определено несколько различных видов переменных. Так, в предыдущих примерах программ использовались переменные, называемые *локальными*, поскольку они объявляются внутри метода.

### Инициализация переменной

Задать значение переменной можно, в частности, с помощью оператора присваивания. Также задать начальное значение переменной можно при ее объявлении. Общая форма инициализации переменной:

*тип имя\_переменной = значение;*

Значение должно соответствовать указанному типу переменной.

Ниже приведены некоторые примеры инициализации переменных.

**int count = 10; // задать начальное значение 10 переменной count.**

**char ch = ‘X’; // инициализировать переменную ch буквенным значением X.**

**float f = 1.2F // переменная f инициализируется числовым значением 1.2.**

Если две или более переменные одного и того же типа объявляются списком, разделяемы запятыми, то этим переменным можно задать, например, начальное значение.

**int a, b = 8, c = 19, d; // инициализировать переменные b и с.**

### Динамическая инициализация

В приведенных выше примерах в качестве инициализаторов переменных использовались только константы, но в C# допускается также динамическая инициализация переменных с помощью любого выражения, действительного на момент объявления переменной. Ниже приведен пример краткой программы для вычисления гипотенузы прямоугольного треугольника по длине его противоположных сторон.

**// Продемонстрировать динамическую инициализацию.**

**using System;**

**class DynInit {**

**static void Main {**

**// Длина сторон прямоугольного треугольника.**

**double s1 = 4.0;**

**double s2 = 5.0;**

**// Инициализировать переменную hypot динамически.**

**double hypot = Math.Sqrt ( (s1 \* s1) + (s2 \* s2) );**

**Console.Write (“Гипотенуза треугольника со сторонами ” +**

**s1 + “ и ” + s2 + “ равна ”);**

**Console.WriteLine (“ {0:#.###}.”, hypot);**

**}**

**}**

Результат выполнения этой программы выглядит следующим образом.

**Гипотенуза треугольника со сторонами 4 и 5 равна 6.403**

В данном примере две локальных переменных инициализируются константами, а третья (hypot) динамически инициализируется вычисляемой длиной гипотенузы. Для такой инициализации используется выражение, указываемое в вызываемом методе **Math.Sqrt()**. Следует особо подчеркнуть, что в выражении для инициализации можно использовать любой элемент, действительный на момент самой инициализации переменной, в том числе вызовы методов, другие переменные или литералы.

### Неявно типизированные переменные

Начиная с версии C# 3.0, компилятору предоставляется возможность самому определить тип локальной переменной, исходя из значения, которым она инициализируется. Такая переменная называется неявно типизированной.

Неявно типизированная переменная объявляется с помощью ключевого слова **var** и должна быть непременно инициализирована. Для определения типа этой переменной компилятору служит тип ее инициализатора, т.е. значения, которым она инициализируется. Рассмотрим такой пример.

**var e = 2.7183;**

В данном примере переменная e инициализируется литералом с плавающей точкой, который по умолчанию имеет тип **double**, и поэтому она относится к типу **double**. Если бы переменная e была объявлена следующим образом:

**var e = 2.7183F;**

то она бы была отнесена к типу **float**.

В приведенном ниже примере программы демонстрируется применение неявно типизированных переменных.

**// Продемонстрировать применение неявно типизированных переменных.**

**using System;**

**class ImplicitlyTypedVar {**

**static void Main {**

**// Эти переменные типизированы неявно. Они отнесены**

**// к типу double, поскольку инициализирующие их**

**// выражения сами относятся к типу double.**

**var s1 = 4.0;**

**var s2 = 5.0;**

**// Переменная hypot типизирована неявно и**

**// относится к типу double, поскольку результат,**

**// возвращаемый методом Math.Sqrt(), имеет тип double.**

**var hypot = Math.Sqrt ( (s1 \* s1) + (s2 \* s2) );**

**Console.Write (“Гипотенуза треугольника со сторонами ” +**

**s1 + “ и ” + s2 + “ равна ”);**

**Console.WriteLine (“ {0:#.###}.”, hypot);**

**// Следующий оператор не может быть скомпилирован,**

**// поскольку переменная s1 имеет тип double и**

**// ей нельзя присвоить десятичное значение.**

**// s1 = 12.2M; // Ошибка!**

**}**

**}**

Результат выполнения этой программы оказывается таким же, как и прежде.

Важно подчеркнуть, что неявно типизированная переменная по-прежнему остается строго типизированной. Нужно обратить внимание на следующую закомментированную строку из приведенной выше программы.

**// s1 = 12.2M; // Ошибка!**

Единственное отличие неявно типизированной переменной от обычной заключается в способе определения ее типа. Как только этот тип будет определен, он закрепляется за переменной до конца ее существования. Это означает, что тип переменной s1 не может быть изменен по ходу выполнения программы.

Неявно типизированные переменные внедрены в C# для особых случаев, и самый примечательный из них имеет отношение к языку интегрированных запросов (LINQ).

И последнее замечание: одновременно можно объявить только одну неявно типизированную переменную. Поэтому объявление

**var s1 = 4.0, s2 = 5.0; // Ошибка!**

является неверным и не может быть скомпилировано. Ведь в нем предпринимается попытка объявить обе переменные, s1 и s2, одновременно.

### Область действия и время существования переменных

Все переменные, использовавшиеся в предыдущих примерах программ, объявлялись в самом начале метода **Main()**. Но в C# локальную переменную разрешается объявлять в любом кодовом блоке. Этот блок и объявляется область действия. Прежде всего область действия определяет видимость имен отдельных элементов, в том числе и переменных, в других частях программы без дополнительного уточнения. Она определяет также время существования локальных переменных.

В C# к числу наиболее важных относятся области действия, определяемые классом и методом. Сейчас же необходимо рассмотреть только те области действия, которые определяются методом или же в самом методе.

Область действия, определяемая методом, начинается открывающей фигурной скобкой и оканчивается закрывающей фигурной скобкой. Но если у этого метода имеются параметры, то они входят в область действия, определяемую данным методом.

Как правило, локальные переменные объявляются в области действия, невидимой для кода, находящегося вне этой области. Поэтому, объявляя переменную в определенной области действия, вы тем самым защищаете ее от доступа или видоизменения вне данной области.

Области действия могут быть вложенными. Например, всякий раз, когда создается кодовый блок, одновременно образуется и новая, вложенная область действия. В этом случае внешняя область действия охватывает внутреннюю область. Это означает, что локальные переменные, объявленные во внешней области действия, будут видимы для кода во внутренней области действия. При этом локальные переменные, объявленные во внутренней области действия, не будут видимы вне этой области.

Для более точного понимания сущности вложенных областей действия рассмотрим следующий пример программы.

**// Продемонстрировать область действия кодового блока.**

**using System;**

**class ScopeDemo {**

**static void Main {**

**int x; // Эта переменная доступна для всего кода внутри метода Main ().**

**x = 10;**

**if (x == 10) { // начать новую область действия**

**int y = 20; // Эта переменная доступна только в данном кодовом блоке.**

**// Здесь доступны обе переменные, x и y.**

**Console.WriteLine (“x и y: ” + x + “ “ +y);**

**x = y \* 2;**

**}**

**// y = 100; // Ошибка! Переменная y здесь недоступна.**

**// А переменная x здесь по-прежнему доступна.**

**Console.WriteLine (“x равно “ +x);**

**}**

**}**

Если в объявление переменной включается инициализатор, то такая переменная инициализируется повторно при каждом входе в тот блок, в котором она объявлена. Рассмотрим следующий пример программы.

**// Продемонстрировать время существования переменной.**

**using System;**

**class VarInitDemo {**

**static void Main () {**

**int x;**

**for (x = 0; x < 3; x++) {**

**int y = -1; // Переменная y инициализируется при каждом входе в блок.**

**Console.WriteLine (“y равно: “ +y); // Здесь всегда выводится -1**

**y = 100;**

**Console.WriteLine (“y теперь равно: ” +y);**

**}**

**}**

**}**

Ниже приведен результат выполнения этой программы.

**y равно: -1**

**y теперь равно: 100**

**y равно: -1**

**y теперь равно: 100**

**y равно: -1**

**y теперь равно: 100**

Как видно, переменная у повторно инициализируется одним и тем же значением -1 при каждом входе во внутренний цикл **for**. И несмотря на то, что после этого цикла ей присваивается значение 100, оно теряется при повторной ее инициализации.

В языке C# имеется еще одна особенность соблюдения правил области действия: несмотря на то, что блоки могут быть вложены, ни у одной из переменных из внутренней области действия не должно быть такое же имя, как и у переменной из внешней области действия. В приведенном ниже примере программы предпринимается попытка объявить две разные переменные с одним и тем же именем, и поэтому программа не может быть скомпилирована.

**/\***

**В этой программе предпринимается попытка объявить во внутренней области действия переменную с таким же самым именем, как и у переменной, определенной во внешней области действия.**

**\*\*\* Эта программа не может быть скомпилирована \*\*\***

**\*/**

**using System;**

**class NestVar {**

**static void Main () {**

**int count;**

**for (count = 0; count < 10; count++) {**

**Console.WriteLine (“Это подсчет: ” +count);**

**int count; // Недопустимо!!!**

**for (count = 0; count < 2; count++)**

**Console.WriteLine (“В этой программе есть ошибка!”);**

**}**

**}**

**}**

Присваивание имен переменным, объявляемым во внутренней области действия, в языках С и C++ неограниченно. Следовательно, в С и C++ объявление переменной **count** в кодовом блоке, входящем во внешний цикл **for**, как в приведенном выше примере, считается вполне допустимым. Но в С и С++ такое объявление одновременно означает сокрытие внешней переменной. Разработчики C# посчитали, что такого рода сокрытие имен может легко привести к программным ошибкам, и поэтому решили запретить его.

### Преобразование и приведение типов

В программировании нередко значения переменных одного типа присваиваются переменным другого типа. Например, в приведенном ниже фрагменте кода целое значение типа int присваивается переменной с плавающей точкой float.

**int i;**

**float f;**

**i = 10;**

**f = i; // присвоить целое значение переменной типа float**

Если в одной операции присваивания смешиваются совместимые типы данных, то значение в правой части оператора присваивания автоматически преобразуется в тип, указанный в левой его части. Поэтому в приведенном выше фрагменте кода значение переменной i сначала преобразуется в тип **float**, а затем присваивается переменной f. Но вследствие строгого контроля типов далеко не все типы данных в C# оказываются полностью совместимыми, а следовательно, не все преобразования типов разрешены в неявном виде. Например, типы **bool** и **int** несовместимы. Правда, преобразование несовместимых типов все-таки может быть осуществлено путем привидения. Приведение типов, по существу, означает явное их преобразование.

### Автоматическое преобразование типов

Когда данные одного типа присваиваются переменной другого типа, *неявное* преобразование типов происходит автоматически при следующих условиях:

* оба типа совместимы;
* диапазон представления чисел целевого типа шире, чем у исходного типа.

Если оба эти условия удовлетворяются, происходит *расширяющее преобразование*. Например, тип **int** достаточно крупный, чтобы вмещать в себя все действительные значения типа **byte**, а кроме того, оба типа, **int** и **byte**, являются совместимыми целочисленными типами, и поэтому для них вполне возможно неявное преобразование.

Числовые типы, как целочисленные, так и с плавающей запятой, вполне совместимы друг с другом для выполнения расширяющих преобразований. Так, приведенная ниже программа составлена совершенно правильно, поскольку преобразование типа **long** в тип **double** является расширяющим и выполняется автоматически.

**// Продемонстрировать неявное преобразование типа long в тип double.**

**using System;**

**class LtoD {**

**static void Main () {**

**long L;**

**double D;**

**L = 100123285L;**

**D = L;**

**Console.WriteLine (“L и D: ” + L + “ ” +D);**

**}**

**}**

Если тип **long** может быть преобразован в тип **double** неявно, то обратное преобразование типа **double** в тип **long** неявным образом невозможно, поскольку оно не является расширяющим. Следовательно, приведенный ниже вариант предыдущей программы составлен неправильно.

**// \*\*\* Эта программа не может быть скомпилирована. \*\*\***

**using System;**

**class LtoD {**

**static void Main () {**

**long L;**

**double D;**

**D = 100123285.0;**

**L = D; // Недопустимо!!!**

**Console.WriteLine (“L и D: ” + L + “ ” +D);**

**}**

**}**

Помимо упомянутых выше ограничений, не допускается неявное взаимное преобразование типов **decimal** и **float** или **double**, а также числовых типов и **char** или **bool**. Кроме того, типы **char** и **bool** несовместимы друг с другом.

### Приведение несовместимых типов

Несмотря на всю полезность неявных преобразований типов, они неспособны удовлетворить все потребности в программировании, поскольку допускают лишь расширяющие преобразования совместимых типов. А во всех остальных случаях приходится обращаться к приведению типов. *Приведение* – это команда компилятору преобразовать результат вычисления выражения в указанный тип. А для этого требуется явное преобразование типов. Общая форма приведения типов:

**(целевой\_тип) выражение**

Рассмотрим для примера следующее объявление переменных.

**double x, y;**

Если результат вычисления выражения x/y должен быть типа **int**, то следует записать следующее:

**(int) (x/y)**

Несмотря на то что переменные x и y относятся к типу **double**, результат вычисления выражения х/у преобразуется в тип **int** благодаря приведению. В данном примере выражение х/у следует непременно указывать в скобках, иначе приведение к типу **int** будет распространяться только на переменную х, а не на результат ее деления на переменную y. Приведение типов в данном случае требуется потому, что неявное преобразование типа **double** в тип **int** невозможно.

Если приведение типов приводит к сужающему преобразованию, то часть информации может быть потеряна. Например, в результате приведения типа **long** к типу **int** часть информации потеряется, если значение типа **long** окажется больше диапазона представления чисел для типа **int**, поскольку старшие разряды этого числового значения отбрасываются. Когда же значение с плавающей точкой приводится к целочисленному, то в результате усечения теряется дробная часть этого числового значения. Так, если присвоить значение 1,23 целочисленной переменной, то в результате в ней останется лишь целая часть исходного числа (1), а дробная его часть (0,23) будет потеряна.

В следующем примере программы демонстрируется ряд преобразований типов, требующих приведения. В этом примере показан также ряд ситуаций, в которых приведение типов становится причиной потери данных.

**// Продемонстрировать приведение типов.**

**using System;**

**class CastDemo {**

**static void Main () {**

**double x, y;**

**byte b;**

**int i;**

**char ch;**

**uint u;**

**short s;**

**long l;**

**x = 10.0;**

**y = 3.0;**

**// Приведение типа double к типу int, дробная часть числа теряется.**

**i = (int) (x / y);**

**Console.WriteLine (“Целочисленный результат деления x/y: ” +i);**

**Console.WriteLine ();**

**// Приведение типа int к типу byte без потери данных.**

**i = 255;**

**b = (byte) i;**

**Console.WriteLine (“b после присваивания 255: ” + b + “ -- без потери данных.”);**

**// Приведение типа int к типу byte с потерей данных.**

**i = 257;**

**b = (byte) i;**

**Console.WriteLine (“b после присваивания 257: ” + b + “ -- с потерей данных.”);**

**Console.WriteLine();**

**// Приведение типа uint к типу short без потери данных.**

**u = 32000;**

**s = (short) u;**

**Console.WriteLine (“s после присваивания 32000: ” + s + “ -- без потери данных.”);**

**// Приведение типа uint к типу short c потерей данных.**

**u = 64000;**

**s = (short) u;**

**Console.WriteLine (“s после присваивания 64000: ” + s + “ -- с потерей данных.”);**

**Console.WriteLine ();**

**// Приведение типа long к типу uint без потери данных.**

**l = 64000;**

**u = (uint) l;**

**Console.WriteLine (“u после присваивания 64000: ” + u + “ -- без потери данных.”);**

**// Приведение типа long к типу uint c потерей данных.**

**l = -12;**

**u = (uint) l;**

**Console.WriteLine (“u после присваивания -12: ” + u + “ -- с потерей данных.”);**

**Console.WriteLine();**

**// Приведение типа int к типу char.**

**b = 88; // код ASCII символа Х**

**ch = (char) b;**

**Console.WriteLine (“ch после присваивания 88: ” + ch);**

**}**

**}**

Вот такой результат дает выполнение этой программы.

**Целочисленный результат деления x/y: 3**

**b после присваивания 255: 255 -- без потери данных.**

**b после присваивания 257: 1 -- с потерей данных.**

**s после присваивания 32000: 32000 -- без потери данных.**

**s после присваивания 64000: -1536 -- с потерей данных.**

**u после присваивания 64000: 64000 -- без потери данных.**

**u после присваивания -12: 4294967284 -- с потерей данных.**

**ch после присваивания 88: X**

### Преобразование типов в выражениях

Помимо операций присваивания, преобразование типов происходит и в самих выражениях. В выражении можно свободно смешивать два или более типа данных, при условии их совместимости друг с другом. Например, в одном выражении допускается применение типов **short** и **ulong**, поскольку оба типа являются числовыми. Когда в выражении смешиваются разные типы данных, они преобразуются в один и тот же тип по порядку следования операций.

Преобразования типов выполняются по принятым в C# правилам продвижения типов. Ниже приведен алгоритм, определяемый этими правилами для операций с двумя операндами.

ЕСЛИ один операнд имеет тип **decimal**, ТО и второй операнд продвигается к типу **decimal** (но если второй операнд имеет тип **float** или **double**, результат будет ошибочным).

ЕСЛИ один операнд имеет тип **double**, ТО и второй операнд продвигается к типу **double**.

ЕСЛИ один операнд имеет тип **float**, ТО и второй операнд продвигается к типу **float**.

ЕСЛИ один операнд имеет тип **ulong**, ТО и второй операнд продвигается к типу **ulong** (но если второй операнд имеет тип **sbyte, short, int** или **long**, результат будет ошибочным).

ЕСЛИ один операнд имеет тип **long**, ТО и второй операнд продвигается к типу **long**.

ЕСЛИ один операнд имеет тип **uint**, а второй – тип **sbyte, short** или **int**, ТО оба операнда продвигаются к типу **long**.

ЕСЛИ один операнд имеет тип **uint**, ТО и второй операнд продвигается к типу **uint**.

ИНАЧЕ оба операнда продвигаются к типу **int**.

Относительно правил продвижения типов необходимо сделать ряд важных замечаний. Во-первых, не все типы могут смешиваться в выражении. В частности, неявное преобразование типа **float** или **double** в тип **decimal** невозможно, как, впрочем, и смешение типа **ulong** с любым целочисленным типом со знаком. Для смешения этих типов требуется явное их приведение.

Во-вторых, особого внимания требует последнее из приведенных выше правил. Оно гласит: если ни одно из предыдущих правил не применяется, то все операнды продвигаются к типу **int**. Следовательно, все значения типа **char, sbyte, byte, ushort** и **short** продвигаются к типу **int** в целях вычисления выражения. Такое продвижение типов называется *целочисленным*. Это также означает, что результат выполнения всех арифметических операций будет иметь тип не ниже **int**.

Следует иметь ввиду, что правила продвижения типов применяются только к значениям, которыми оперируют при вычислении выражения. Так, если значение переменной типа **byte** продвигается к типу **int** внутри выражения, то вне выражения эта переменная по-прежнему относится к типу **byte**. Продвижение типов затрагивает только вычисление выражения.

Но продвижение типов может иногда привести к неожиданным результатам. Если, например, в арифметической операции используются два значения типа **byte**, то происходит следующее. Сначала операнды типа **byte** продвигаются к типу **int**. А затем выполняется операция, дающая результат типа **int**. Следовательно, результат выполнения операции, в которой участвуют два значения типа **byte**, будет иметь тип **int**. Но ведь это не тот результат, который можно было бы с очевидностью предположить. Рассмотрим следующий пример программы.

**// Пример неожиданного результата продвижения типов!**

**using System;**

**class PromDemo {**

**static void Main () {**

**byte b;**

**b = 10;**

**b = (byte) (b \* b); // Необходимо приведение типов!!!**

**Console.WriteLine (“b: ” +b);**

**}**

**}**

Как ни странно, но когда результат вычисления выражения b \* b присваивается обратно переменной b, то возникает потребность в приведении к типу **byte**! Объясняется это тем, что в выражении b \* b значение переменной b продвигается к типу **int** и поэтому не может быть присвоено переменной типа **byte** без приведения типов. Нужно иметь это обстоятельство в виду, если получите неожиданное сообщение об ошибке несовместимости типов в выражениях, которые, на первый взгляд, кажутся совершенно правильными.

Аналогичная ситуация возникает при выполнении операций с символьными операндами. Например, в следующем фрагменте кода требуется обратное приведение к типу **char**, поскольку операнды ch1 и ch2 в выражении продвигаются к типу **int**.

**char ch1 = ‘a’, ch2 = ‘b’;**

**ch1 = (char) (ch1 + ch2);**

Без приведения типов результат сложения операндов ch1 и ch2 будет иметь тип **int**, и поэтому его нельзя присвоить переменной типа **char**.

Продвижение типов происходит и при выполнении унарных операций, например, с унарным минусом. Операнды унарных операций более мелкого типа, чем **int (byte, sbyte, short** и **ushort)**, т.е. с более узким диапазоном представления чисел, продвигаются к типу **int**. То же самое происходит с операндом типа **char**. Кроме того, если выполняется унарная операция отрицания значения типа **uint**, то результат продвигается к типу **long**.

### Приведение типов в выражениях

Приведение типов можно применять и к отдельным частям крупного выражения. Это позволяет точнее управлять преобразованиями типов при вычислении выражения. Рассмотрим следующий пример программы, в которой выводятся квадратные корни от 1 до 10 и отдельно целые и дробные части каждого числового результата. Для этого в данной программе применяется приведение типов, благодаря которому результат, возвращаемый методом **Math.Sqrt()**, преобразуется в тип **int**.

**// Пример приведения типов в выражениях.**

**using System;**

**class CastExpr {**

**static void Main () {**

**double n;**

**for (n = 1.0; n <= 10; n++) {**

**Console.WriteLine (“Квадратный корень из {0} равен {1}”, n, Math.Sqrt(n) );**

**Console.WriteLine (“Целая часть числа: {0}”, (int) Math.Sqrt(n) );**

**Console.WriteLine (“Дробная часть числа: {0}”, Math.Sqrt(n) – (int) Math.Sqrt(n) );**

**Console.WriteLine ();**

**}**

**}**

**}**

Вот как выглядит часть результата выполнения этой программы.

**Квадратный корень из 1 равен 1**

**Целая часть числа: 1**

**Дробная часть числа: 0**

**Квадратный корень из 2 равен 1.4142135623731**

**Целая часть числа: 1**

**Дробная часть числа: 0.414213562373095**

**Квадратный корень из 3 равен 1.73205080756888**

**Целая часть числа: 1**

**Дробная часть числа: 0.732050807568877**

Как видно, приведение результата, возвращаемого методом **Math.Sqrt()**, к типу **int** позволяет получить целую часть числа. Так, в выражении

**Math.Sqrt(n) – (int) Math.Sqrt(n)**

приведение к типу **int** дает целую часть числа, которая затем вычитается из всего числа, а в итоге получается дробная его часть. Следовательно, результат вычисления данного выражения имеет тип **double**. Но к типу **int** приводится только значение, возвращаемое вторым методом **Math.Sqrt()**.

### Арифметические операторы

Операторы +, -, \* и / действуют так, как предполагает их обозначение. Их можно применять к любому встроенному числовому типу данных.

Не следует забывать, что когда оператор / применяется в целому числу, то любой остаток от деления отбрасывается; например, результат целочисленного деления 10/3 будет равен 3. Остаток от этого деления можно получить с помощью оператора деления по модулю (%), который иначе называется *оператором вычисления остатка*. Он дает остаток от целочисленного деления. Например, 10 % 3 равно 1. В C# оператор % можно применять как к целочисленным типам данных, так и к типам с плавающей точкой. Поэтому 10.0 % 3 также равно 1. В этом отношении C# отличается от языков C и C++, где операции деления по модулю разрешаются только для целочисленных типов данных. В приведенном ниже примере программы демонстрируется применение оператора деления по модулю.

**// Продемонстрировать применение оператора %.**

**using System;**

**class ModDemo {**

**static void Main () {**

**int iresult, irem;**

**double dresult, drem;**

**iresult = 10 /3;**

**irem = 10 % 3;**

**dresult = 10.0 / 3.0;**

**drem = 10.0 % 3.0;**

**Console.WriteLine (“Результат и остаток деления 10 / 3: ” + iresult + “ ” + irem);**

**Console.WriteLine (“Результат и остаток деления 10.0 / 3.0: ” + dresult + “ ” +drem);**

**}**

**}**

Результат выполнения этой программы приведен ниже.

**Результат и остаток деления 10 / 3: 3 1**

**Результат и остаток деления 10.0 / 3.0: 3.33333333333333 1**

Как видно, обе операции % целочисленного типа и с плавающей точкой, дают один и тот же остаток, равный 1.

### Операторы инкремента и декремента

Операторы инкремента (++) и декремента (--) обладают рядом особых и довольно интересных свойств. Но сначала выясним основное назначение этих операторов.

Оператор инкремента увеличивает свой операнд на 1, а оператор декремента уменьшает операнд на 1. Следовательно, оператор

x++; равнозначен оператору x = x + 1;

а оператор

x--; равнозначен оператору x = x – 1;

Следует, однако, иметь ввиду, что в инкрементной или декрементной форме значение переменной x вычисляется только один раз, а не два раза. В некоторых случаях это позволяет повысить эффективность выполнения программы.

Оба оператора инкремента и декремента можно указывать до операнда (в префиксной форме) или же после операнда (в постфиксной форме). Например, оператор

x = x + 1;

может быть записан в следующем виде:

**++x; // префиксная форма**

Или же в таком виде:

**x++; // постфиксная форма**

В приведенном выше примере форма инкремента (префиксная или постфиксная) особого значения не имеет. Но если оператор инкремента или декремента используется в длинном выражении, то отличие в форме его записи уже имеет значение. Когда оператор инкремента или декремента предшествует своему операнду, то результатом операции становится значение операнда после инкремента или декремента. А когда оператор инкремента или декремента следует после своего операнда, то результатом операции становится значение операнда до инкремента или декремента. Рассмотрим следующий фрагмент кода.

**x = 10;**

**y = ++x;**

В данном случае значение переменной у будет установлено равным 11, поскольку значение переменной х сначала увеличивается на 1, а затем присваивается переменной y. Но во фрагменте кода

**x = 10;**

**y = x++;**

значение переменной у будет установлено равным 10, так как в этом случае значение переменной х сначала присваивается переменной y, а затем увеличивается на 1. В обоих случаях значение переменной х оказывается равным 11. Отличие состоит лишь в том, когда именно это значение станет равным 11: до или после его присваивания переменной y.

Возможность управлять моментом инкремента или декремента дает немало преимуществ при программировании. Обратимся к следующему примеру программы, в которой формируется последовательный ряд чисел.

**// Продемонстрировать отличие между префиксной**

**// и постфиксной формами оператора инкремента (++).**

**using System;**

**class PrePostDemo {**

**static void Main () {**

**int x, y;**

**int i;**

**x = 1;**

**y = 0;**

**Console.WriteLine (“Ряд чисел, полученных ” +**

**“с помощью оператора y = y + x++;”);**

**for (i = 0; i < 10; i++) {**

**y = y + x++; // постфиксная форма оператора ++**

**Console.WriteLine (y + “ ”);**

**}**

**Console.WriteLine();**

**x = 1;**

**y = 0;**

**Console.WriteLine (“Ряд чисел, полученных ” +**

**“с помощью оператора y = y+ ++x;”);**

**for (i = 0; i < 10; i++) {**

**y = y + ++x; // префиксная форма оператора ++**

**Console.WriteLine (y+ “ ”);**

**}**

**Console.WriteLine();**

**}**

**}**

Выполнение этой программы дает следующий результат.

**Ряд чисел, полученных с помощью оператора y = y + x++;**

**1**

**3**

**6**

**10**

**15**

**21**

**28**

**36**

**45**

**55**

**Ряд чисел, полученных с помощью оператора y = y + ++x;**

**2**

**5**

**9**

**14**

**20**

**27**

**35**

**44**

**54**

**65**

Как подтверждает приведенный выше результат, в операторе

**y = y + x++;**

первоначальное значение переменной х складывается с самим собой, а полученный результат присваивается переменной y при следующей итерации. После этого значение переменной x увеличивается на 1. Но в операторе

**y = y + ++x**

значение переменной x сначала увеличивается на 1, затем складывается с первоначальным значением этой же переменной, а полученный результат присваивается переменной y при следующей итерации. Как следует из приведенного выше результата, простая замена префиксной формы записи оператора ++x постфиксной формой х++ приводит к существенному изменению последовательного ряда получаемых чисел.

Не нужно пугаться выражений, подобных следующему:

y + ++x

Такое расположение рядом двух операторов может показаться не совсем привычным, но компилятор воспримет их в правильной последовательности. Нужно лишь запомнить, что в данном выражении значение переменной y складывается с увеличенным на 1 значением переменной x.

### Операторы отношения и логические операторы

В обозначениях *оператор отношения* и *логический оператор* термин *отношения* означает взаимосвязь, которая может существовать между двумя значениями, а термин *логический* – взаимосвязь между логическими значениями «истина» и «ложь». И поскольку операторы отношения дают истинные и ложные результаты, то они нередко применяются вместе с логическими операторами. Именно по этой причине они и рассматриваются совместно в данном разделе.

Ниже перечислены операторы отношения.

**== Равно**

**!= Не равно**

**> Больше**

**< Меньше**

**>= Больше или равно**

**<= Меньше или равно**

К числу логических относятся операторы, приведенные ниже.

**& И**

**| ИЛИ**

**^ Исключающее ИЛИ**

**&& Укороченное И**

**|| Укороченное ИЛИ**

**! НЕ**

Результатом выполнения оператора отношения или логического оператора является логическое значение типа **bool**.

В целом, объекты можно сравнивать на равенство или неравенство, используя операторы отношения == и !=. А операторы сравнения <, >, <= или >= могут применяться только к тем типам данных, которые поддерживают отношение порядка. Следовательно, операторы отношения можно применять ко всем числовым типам данных. Но значения типа **bool** могут сравниваться только на равенство или неравенство, поскольку истинные (true) и ложные (false) значения не упорядочиваются. К примеру, сравнение true > false в C# не имеет смысла.

Операнды логических операторов должны относиться к типу **bool**, а результат выполнения логической операции также относится к типу **bool**. Логические операторы &, |, ^ и ! поддерживают основные логические операции И, ИЛИ, исключающее ИЛИ, и НЕ в соответствии с приведенной ниже таблицей истинности.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** | **!p** |
| false | false | false | false | false | true |
| true | false | false | true | true | false |
| false | true | false | true | true | true |
| true | true | true | false | false | false |

Как следует из приведенной выше таблицы, результатом выполнения логической операции исключающее ИЛИ будет истинное значение (true), если один и только один ее операнд имеет значение true.

Ниже приведен пример программы, демонстрирующей применение нескольких операторов отношения и логических операторов.

**// Продемонстрировать применение операторов**

**// отношения и логических операторов.**

**using System;**

**class RelLogOps {**

**static void Main () {**

**int i, j;**

**bool b1, b2;**

**i = 10;**

**j = 11;**

**if (i < j) Console.WriteLine (“i < j”);**

**if (i <= j) Console.WriteLine (“i <= j”);**

**if (i != j) Console.WriteLine (“i != j”);**

**if (i == j) Console.WriteLine (“Нельзя выполнить”);**

**if (i >= j) Console.WriteLine (“Нельзя выполнить”);**

**if (i > j) Console.WriteLine (“Нельзя выполнить”);**

**b1 = true;**

**b2 = false;**

**if (b1 & b2) Console.WriteLine (“Нельзя выполнить”);**

**if (! (b1 & b2) ) Console.WriteLine (“! (b1 & b2) – true”);**

**if (b1 | b2) Console.WriteLine (“b1 | b2 – true”);**

**if (b1 ^ b2) Console.WriteLine (“b1 ^ b2 – true”);**

**}**

**}**

Выполнение этой программы дает следующий результат.

**i < j**

**i <= j**

**i != j**

**! (b1 & b2) – true**

**b1 | b2 – true**

**b1 ^ b2 – true**

Ряд операций в C# выполняются по правилам формальной логики. Эти логические операции могут быть построены с помощью логических операторов, поддерживаемых в C#. Следовательно, в C# предусмотрен такой набор логических операторов, которого достаточно для построения практически любой логической операции, в том числе импликации. *Импликация* – это двоичная операция, результатом которой является ложное значение только в том случае, если левый ее операнд имеет истинное значение, а правый – ложное. (Операция импликации отражает следующий принцип: истина не может подразумевать ложь). Ниже приведена таблица истинности для операции импликации.

|  |  |  |
| --- | --- | --- |
| **p** | **q** | **Результат импликации p и q** |
| true | true | true |
| true | false | false |
| false | false | true |
| false | true | true |

Операция импликации может быть построена на основе комбинации логических операторов ! и |, как в приведенной ниже строке кода.

**!p | q**

В следующем примере программы демонстрируется подобная реализация операции импликации.

**// Построение операции импликации в C#.**

**using System;**

**class Implication {**

**static void Main () {**

**bool p = false, q = false;**

**int i, j;**

**for (i = 0; i < 2; i++) {**

**for (j = 0; j <2; j++) {**

**if (i == 0) p = true;**

**if (i == 1) p = false;**

**if (j == 0) q = true;**

**if (j == 1) q = false;**

**Console.WriteLine (“p равно” + p + “, q равно” +q);**

**if (!p | q)**

**Console.WriteLine (“Результат импликации ” + p + “ и “ + q + “равен “ +true);**

**Console.WriteLine ();**

**}**

**}**

**}**

**}**

Результат выполнения этой программы выглядит так.

**p равно True, q равно True**

**Результат импликации True и True равен True**

**p равно True, q равно False**

**p равно False, q равно True**

**Результат импликации False и True равен True**

**p равно False, q равно False**

**Результат импликации False и False равен True**

### Укороченные логические операторы

В C# предусмотрены также специальные, *укороченные*, варианты логических операторов И и ИЛИ, предназначенные для получения более эффективного кода. Поясним это на следующих примерах логических операций. Если первый операнд логической операции И имеет ложное значение (false), то ее результат будет иметь ложное значение независимо от значения второго операнда. Если же первый операнд логической операции ИЛИ имеет истинное значение (true), то ее результат будет иметь истинное значение независимо от значения второго операнда. Благодаря тому что значение второго операнда в этих операциях вычислять не нужно, экономится время и повышается эффективность кода.

Укороченная логическая операция И выполняется с помощью оператора &&, а укороченная логическая операция ИЛИ – с помощью оператора ||. Единственное отличие укороченного логического оператора от обычного заключается в том, что второй его операнд вычисляется только по мере необходимости.

В приведенном ниже примере демонстрируется применение укороченного логического оператора И. В этой программе с помощью операции деления по модулю определяется следующее: делится ли значение переменной d на значение переменной n нацело. Если остаток от деления n/d равен нулю, то n делится на d нацело. Но поскольку данная операция подразумевает деление, то для проверки условия деления на нуль служит укороченный логический оператор И.

**// Продемонстрировать применение укороченных логических операторов.**

**using System;**

**class SCops {**

**static void Main () {**

**int n, d;**

**n = 10;**

**d = 2;**

**if (d !=0 && (n % d) == 0)**

**Console.WriteLine (n+ “ делится нацело на ” +d);**

**d = 0; // задать нулевое значение переменной d**

**// d равно нулю, поэтому второй операнд не вычисляется**

**if (d !=0 && (n % d) == 0)**

**Console.WriteLine (n +“ делится нацело на ” +d);**

**// Если теперь попытаться сделать то же самое без укороченного**

**// логического оператора, то возникнет ошибка из-за деления на нуль.**

**if (d !=0 & (n % d) == 0)**

**Console.WriteLine (n + “ делится нацело на ” +d);**

**}**

**}**

Для исключения ошибки из-за деления на нуль в операторе **if** сначала проверяется условие: равно ли нулю значение переменной d. Если оно равно нулю, то на этом выполнение укороченного логического оператора И завершается, а последующая операция деления по модулю не выполняется. Так, при первой проверке значение переменной d оказывается равным 2, поэтому выполняется операция деления по модулю. А при второй проверке это значение оказывается равным нулю, следовательно, операция деления по модулю пропускается, чтобы исключить деление на нуль. И, наконец, выполняется обычный логический оператор И, когда вычисляются оба операнда. Если при этом происходит деление на нуль, то возникает ошибка при выполнении.

Укороченные логические операторы иногда оказываются более эффективными, чем их обычные аналоги. Дело в том, что в некоторых случаях требуется вычислять оба операнда логической операции И либо ИЛИ из-за возникающих побочных эффектов. Рассмотрим следующий пример программы.

**// Продемонстрировать значение побочных эффектов.**

**class SideEffects {**

**static void Main () {**

**int i;**

**bool someCondition = false;**

**i = 0;**

**// Значение переменной i инкрементируется,**

**// несмотря на то, что оператор if не выполняется.**

**if (someCondition & (++i < 100))**

**Console.WriteLine (“Не выводится”);**

**Console.WriteLine (“Оператор if выполняется: ” +i); // выводится 1**

**// В данном случае значение переменной i не инкрементируется,**

**// поскольку инкремент в укороченном логическом операторе опускается.**

**if (Somecondition && (++i < 100))**

**Console.WriteLine (“Не выводится”);**

**Console.WriteLine (“Оператор if выполняется: ” +i); // по-прежнему 1 !!**

**}**

**}**

Прежде всего обратим внимание на то, что переменная someCondition типа **bool** инициализируется значением false. Далее проанализируем каждый оператор **if**. В первом операторе **if** переменная i инкрементируется, несмотря на то что значение переменной someCondition равно false. Когда применяется логический оператор &, выражение в правой части этого оператора вычисляется независимо от значения выражения в его левой части. А во втором оператора **if** применяется укороченный логический оператор. В этом случае значение переменной i не инкрементируется, поскольку левый операнд имеет значение false, следовательно, выражение в правой части данного оператора пропускается. Из этого следует вывод: если в коде предполагается вычисление правого операнда логической операции И либо ИЛИ, то необходимо пользоваться неукороченными формами логических операций, доступных в C#.

И последнее замечание: укороченный оператор И называется также *условным логическим оператором И*, а укороченный оператор ИЛИ – *условным логическим оператором ИЛИ.*

### Оператор присваивания

*Оператор присваивания* обозначается одиночным знаком равенства (=). Ниже приведена его общая форма.

**имя\_переменной = выражение**

Здесь имя\_переменной должно быть совместимо с типом выражения.

У оператора присваивания имеется одна интересная особенность: он позволяет создавать цепочку операций присваивания. Рассмотрим, например, следующий фрагмент кода.

**int x, y, z;**

**x = y = z = 100; // присвоить значение 100 переменным x, y и z**

Это значение присваивается сначала переменной z, затем переменной y, а после переменной x. Такой способ присваивания «по цепочке» удобен для задания общего значения целой группе переменных.

### Составные операторы присваивания

В C# предусмотрены специальные составные операторы присваивания, упрощающие программирование некоторых операций присваивания. Обратимся сначала к простому примеру. Приведенный ниже оператор присваивания

**x = x + 10;**

можно переписать, используя следующий составной оператор присваивания.

**x += 10;**

Пара операторов += указывает компилятору на то, что переменной x должно быть присвоено ее первоначальное значение, увеличенное на 10.

Рассмотрим еще один пример. Оператор

**x = x – 100;**

и оператор

**x -= 100;**

выполняют одни и те же действия.

Для многих двоичных операций, т.е. операций, требующих наличия двух операндов, существуют отдельные составные операторы присваивания. Общая форма всех этих операторов имеет следующий вид.

**имя\_переменной ор = выражение**

где ор – арифметический или логический оператор, применяемый вместе с оператором присваивания.

Ниже перечислены составные операторы присваивания для арифметических и логических операций.

|  |  |  |  |
| --- | --- | --- | --- |
| **+=** | **-=** | **\*=** | **/=** |
| **%=** | **&=** | **|=** | **^=** |

Составные операторы присваивания записываются более кратко, чем их несоставные эквиваленты. Поэтому их иногда еще называют *укороченными операторами присваивания.*

Помимо компактности, составные операторы присваивания дают более эффективный исполняемый код, поскольку левый операнд этих операторов вычисляется только один раз. Именно по этим причинам составные операторы присваивания чаще всего применяются в программах, профессионально написанных на C#.

### Поразрядные операторы

В С# предусмотрен ряд *поразрядных* операторов, расширяющих круг задач, для решения которых можно применять C#. Поразрядные операторы воздействуют на отдельные двоичные разряды (биты) своих операндов. Они определены только для целочисленных операндов, поэтому их нельзя применять к данным типа **bool, float** и **double**.

Эти операторы называются *поразрядными*, поскольку они служат для проверки, установки или сдвига двоичных разрядов, составляющих целое значение. Среди прочего поразрядные операторы применяются для решения самых разных задач программирования на уровне системы, включая, например, анализ информации состояния устройства. Все доступные поразрядные операторы приведены ниже.

**& Поразрядное И**

**| Поразрядное ИЛИ**

**^ Поразрядное исключающее ИЛИ**

**>> Сдвиг вправо**

**<< Сдвиг влево**

**~ Дополнение до 1 (унарный оператор НЕ)**

### Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ

Поразрядные операторы И, ИЛИ, исключающее ИЛИ и НЕ обозначаются следующим образом: &, |, ^ и ~. Они выполняют те же функции, что и их логические аналоги, рассмотренные выше. Но в отличие от логических операторов, поразрядные операторы действуют на уровне отдельных двоичных разрядов. Ниже приведены результаты поразрядных операций с двоичными единицами и нулями.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **p** | **q** | **p & q** | **p | q** | **p ^ q** | **~p** |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

С точки зрения наиболее распространенного применения поразрядную операцию И можно рассматривать как способ подавления отдельных двоичных разрядов. Это означает, что если какой-нибудь бит в любом из операндов равен 0, то соответствующий бит результата будет сброшен в 0. Например:

1101 0011

1010 1010

&\_\_\_\_\_\_\_\_

1000 0010

В приведенном ниже примере программы демонстрируется применение поразрядного оператора & для преобразования нечетных чисел в четные. Для этой цели достаточно сбросить младший разряд числа. Например, число 9 имеет следующий двоичный вид: 0000 1001. Если сбросить младший разряд этого числа, то оно станет числом 8, а в двоичной форме – 0000 1001.

**// Применить поразрядный оператор И, чтобы сделать число четным.**

**using System;**

**class MakeEven {**

**static void Main () {**

**ushort num;**

**ushort i;**

**for (i = 1; i <= 10; i++) {**

**num = i;**

**Console.WriteLine (“num: ” +num);**

**num = (ushort) (num & 0xFFFE);**

**Console.WriteLine (“num после сброса младшего разряда: ” + num + “\n”);**

**}**

**}**

**}**

Результат выполнения этой программы приведен ниже.

**num: 1**

**num после сброса младшего разряда: 0**

**num: 2**

**num после сброса младшего разряда: 2**

**num: 3**

**num после сброса младшего разряда: 2**

**num: 4**

**num после сброса младшего разряда: 4**

**num: 5**

**num после сброса младшего разряда: 4**

**num: 6**

**num после сброса младшего разряда: 6**

**num: 7**

**num после сброса младшего разряда: 6**

**num: 8**

**num после сброса младшего разряда: 8**

**num: 9**

**num после сброса младшего разряда: 8**

**num: 10**

**num после сброса младшего разряда: 10**

Шестнадцатеричное значение 0xFFFE, используемое в поразрядном операторе И, имеет следующую двоичную форму: 1111 1111 1111 1110. Таким образом, поразрядная операция И оставляет без изменения все двоичные разряды в числовом значении переменной num, кроме младшего разряда, который сбрасывается в нуль. В итоге четные числа не претерпевают никаких изменений, а нечетные уменьшаются на 1 и становятся четными.

Поразрядным оператором И удобно также пользоваться для определения установленного или сброшенного состояния отдельного двоичного разряда. В следующем примере программы определяется, является ли число нечетным.

**// Применить поразрядный оператор И, чтобы определить,**

**// является ли число нечетным.**

**using System;**

**class IsOdd {**

**static void Main () {**

**ushort num;**

**num = 10;**

**if ((num & 1) == 1)**

**Console.WriteLine (“Не выводится.”);**

**num = 11;**

**if ((num & 1) == 1)**

**Console.WriteLine (num + “ – нечетное число.”);**

**}**

**}**

Вот как выглядит результат выполнения этой программы.

**11 – нечетное число.**

В обоих операторах **if** из приведенной выше программы выполняется поразрядная операция И на числовыми значениями переменной num и 1. Если младший двоичный разряд числового значения переменной num установлен, то есть содержит двоичную единицу, то результат поразрядной операции num & 1 оказывается равным 1. В противном случае он равен нулю. Поэтому оператор **if** может быть выполнен успешно лишь в том случае, если проверяемое число оказывается нечетным.

Возможностью проверять состояние отдельных двоичных разрядов с помощью поразрядного оператора & можно воспользоваться для написания программы, в которой отдельные двоичные разряды проверяемого значения типа **byte** приводятся в двоичной форме. Ниже показан один из способов написания такой программы.

**// Показать биты, составляющие байт.**

**using System;**

**class ShowBite {**

**static void Main () {**

**int t;**

**byte val;**

**val = 123;**

**for (t = 128; t > 0; t = t / 2) {**

**if ((val & t) !=0) Console.Write (“1 ”);**

**if ((val & t) == 0) Console.Write (“0 ”);**

**}**

**}**

**}**

Выполнение этой программы дает следующий результат.

**01111011**

В цикле **for** из приведенной выше программы каждый бит значения переменной val проверяется с помощью поразрядного оператора И, чтобы выяснить, установлен ли этот бит или сброшен. Если он установлен, то выводится цифра 1, а если сброшен, то выводится цифра 0.

Поразрядный оператор ИЛИ может быть использован для установки отдельных двоичных разрядов. Если в 1 установлен какой-нибудь бит в любом из операндов этого оператора, то в 1 будет установлен и соответствующий бит в другом операнде.

1101 0011

1010 1010

|\_\_\_\_\_\_\_\_

1111 1011

Используя поразрядный оператор ИЛИ, можно без особого труда превратить упоминавшийся выше пример программы, преобразующий нечетные числа в четные, в приведенный ниже обратный пример, где четные числа преобразуются в нечетные.

**// Применить поразрядный оператор ИЛИ, чтобы сделать число нечетным.**

**using System;**

**class MakeOdd {**

**static void Main () {**

**ushort num;**

**ushort i;**

**for (i = 1; i < 10; i++) {**

**num = i;**

**Console.WriteLine (“num: ” + num);**

**num = (ushort) (num | 1);**

**Console.WriteLine (“num после установки младшего разряда: ” + num + “\n”);**

**}**

**}**

**}**

Результат выполнения этой программы выглядит следующим образом.

**num: 1**

**num после сброса младшего разряда: 1**

**num: 2**

**num после сброса младшего разряда: 3**

**num: 3**

**num после сброса младшего разряда: 3**

**num: 4**

**num после сброса младшего разряда: 5**

**num: 5**

**num после сброса младшего разряда: 5**

**num: 6**

**num после сброса младшего разряда: 7**

**num: 7**

**num после сброса младшего разряда: 7**

**num: 8**

**num после сброса младшего разряда: 9**

**num: 9**

**num после сброса младшего разряда: 9**

**num: 10**

**num после сброса младшего разряда: 11**

В приведенной выше программе выполняется поразрядная операция ИЛИ над каждым числовым значением переменной num и 1, поскольку 1 дает двоичное значение, в котором установлен младший разряд. В результате поразрядный операции ИЛИ над 1 и любым другим значением младший разряд последнего устанавливается, тогда как все остальные разряды остаются без изменения. Таким образом, результирующее числовое значение получается нечетным, если исходное значение было четным.

Поразрядный оператор исключающее ИЛИ устанавливает двоичный разряд операнда в том и только в том случае, если двоичные разряды сравниваемых операндов оказываются разными, как в приведенном ниже примере.

0111 1111

1011 1001

^\_\_\_\_\_\_\_\_

1100 0110

У поразрядного оператора исключающее ИЛИ имеет одно интересное свойство, которое оказывается полезным в самых разных ситуациях. Так, если выполнить сначала поразрядную операцию исключающее ИЛИ одного значения Х с другим значением Y, а затем такую же операцию над результатом предыдущей операции и значением Y, то вновь получится первоначальное значение X. Это означает, что в приведенном ниже фрагменте кода

**R1 = X ^ Y;**

**R2 = R1 ^ Y;**

Значение переменной R2 оказывается в итоге таким же, как и значение переменной X. Следовательно, в результате двух последовательно выполняемых поразрядных операций исключающее ИЛИ, в которых используется одно и то же значение, получается первоначальное значение. Этим свойством данной операции можно воспользоваться для написания простой программы шифрования, в которой некоторое целое значение служит в качестве ключа для кодирования и декодирования сообщения с помощью операции исключающее ИЛИ над символами этого сообщения. В первый раз операция исключающее ИЛИ выполняется для кодирования открытого текста в зашифрованный, а второй раз – для декодирования зашифрованного текста в открытый. Разумеется, такое шифрование не представляет никакой практической ценности, поскольку оно может быть легко разгадано. Тем не менее, оно служит интересным примером для демонстрации результатов применения поразрядных операторов исключающее ИЛИ, как в приведенной ниже программе.

**// Продемонстрировать применение поразрядного оператора исключающее ИЛИ.**

**using System;**

**class Encode {**

**static void Main () {**

**char ch1 = ‘H’;**

**char ch2 = ‘i’;**

**char ch 3 = ‘!’;**

**int key = 88;**

**Console.WriteLine (“Исходное сообщение: ” + ch1 + ch2 + ch3);**

**// Зашифровать сообщение.**

**ch1 = (char) (ch1 ^ key);**

**ch2 = (char) (ch2 ^ key);**

**ch3 = (char) (ch3 ^ key);**

**Console.WriteLine (“Зашифрованное сообщение: ” + ch1 + ch2 + ch3);**

**// Расшифровать сообщение.**

**ch1 = (char) (ch1 ^ key);**

**ch2 = (char) (ch2 ^ key);**

**ch3 = (char) (ch3 ^ key);**

**Console.WriteLine (“Расшифрованное сообщение: ” + ch1 + ch2 + ch3);**

**}**

**}**

Ниже приведен результат выполнения этой программы.

**Исходное сообщение: Hi!**

**Зашифрованное сообщение: □ly;**

**Расшифрованное сообщение: Hi!**

Как видно, в результате выполнения двух последовательностей поразрядных операций исключающее ИЛИ получается расшифрованное сообщение.

Поразрядный унарный оператор НЕ (или оператор дополнения до 1) изменяет на обратное состояние всех двоичных разрядов операнда. Так, если некоторое целое значение А имеет комбинацию двоичных разрядов 1001 0110, то в результате поразрядной операции ~А получается значение с комбинацией двоичных разрядов 0110 1001.

В следующем примере программы демонстрируется применение поразрядного оператора НЕ с выводом некоторого числа и его дополнения до 1 в двоичном коде.

**// Продемонстрировать применение поразрядного унарного оператора НЕ.**

**using System;**

**class NotDemo {**

**static void Main () {**

**sbyte b = -34;**

**for (int t = 128; t > 0; t = t/2) {**

**if ((b & t) != 0) Console.Write (“1 “);**

**if ((b & t) = 0) Console.Write (“0 ”);**

**}**

**Console.WriteLine ();**

**// обратить все биты.**

**b = (sbyte) ~b;**

**for (int t = 128; t > 0; t = t/2) {**

**if ((b & t) != 0) Console.Write (“1 ”);**

**if ((b & t) == 0) Console.Write (“0 ”);**

**}**

**}**

**}**

Результат выполнения этой программы приведен ниже.

**1 1 0 1 1 1 1 0**

**0 0 1 0 0 0 0 1**

### Операторы сдвига

Двоичные разряды соответствуют форме представления чисел в степени 2, и поэтому операторы сдвига могут быть использованы для умножения или деления целых значений на 2. Так, при сдвиге вправо целое значение удваивается, а при сдвиге влево – уменьшается наполовину. Разумеется, все это справедливо лишь в том случае, если крайние разряды не теряются при сдвиге в ту или иную сторону.

int n = 10;

n = n >> 1;

Результат выполнения этой операции: 5

int n = 10;

n = n << 1;

Результат выполнения этой операции: 20

## Введение в классы, объекты и методы

### Общая форма определения класса

При определении класса объявляются данные, которые он содержит, а также код, оперирующий этими данными. Если самые простые классы могут содержать только код или только данные, то большинство настоящих классов содержит и то и другое.

Вообще говоря, данные содержатся в *членах данных*, определяемых классом, а код – в *функциях-членах*. В C# предусмотрено несколько разновидностей членов данных и функций-членов. Например, к членам данных, называемым также *полями*, относятся переменные экземпляра и статические переменные, а к функциям-членам – методы, конструкторы, деструкторы, индексаторы, события, операторы и свойства. Ограничимся пока что рассмотрением самых основных основных компонентов класса: переменных экземпляра и методов.

Класс создается с помощью ключевого слова **class**. Общая форма определения простого класса, содержащая только переменные экземпляра и методы:

class имя\_класса

{

// Объявление переменных экземпляра.

доступ тип переменная 1;

доступ тип переменная 2;

//...

доступ тип переменная N;

// Объявление методов.

доступ возвращаемый\_тип метод1 (параметры) {

// тело метода

}

доступ возвращаемый\_тип метод2 (параметры) {

// тело метода

}

//...

доступ возвращаемый\_тип методN (параметры) {

// тело метода

}

}

Нужно обратить внимание на то, что перед каждым объявлением переменной и метода указывается *доступ*. Это спецификатор доступа, например, **public**, определяющий порядок доступа к данному члену класса. Члены класса могут быть как закрытыми (**private**) в пределах класса, так и открытыми (**public**), т.е. более доступными. Спецификатор доступа определяет *тип* разрешенного доступа. Указывать спецификатор доступа не обязательно, но если он отсутствует, то объявляемый член считается закрытым в пределах класса. Члены с закрытым доступом могут использоваться только другими членами их класса. В примерах программ, приведенных в этой главе, все члены, за исключением метода **Main()**, обозначаются как открытые (**public**). Это означает, что их можно использовать во всех остальных фрагментах кода – даже в тех, что определены за пределами класса.

*Помимо спецификаторов доступа, в объявлении члена класса могут также присутствовать один или несколько модификаторов.*

Правильно сконструированный класс должен определять одну и только одну логическую сущность. Например, класс, в котором хранятся Ф.И.О. и номера телефонов, обычно не содержит сведения о фондовом рынке, среднем уровне осадков и т.д.

### Определение класса

Создадим класс **Building**, в котором будут храниться три элемента информации о зданиях: количество этажей, общая площадь и количество жильцов.

Ниже приведен первый вариант данного класса. В нем определены три переменные экземпляра: **Floors, Arena и Occupants**. Сейчас в данном классе отсутствуют методы. Это означает, что в данный момент класс состоит только из данных.

class Building

{

public int Floors; // количество этажей

public int Area; // общая площадь здания

public int Occupants; // количество жильцов

}

Переменная доступа объявляется также как, как и локальная переменная. Все эти переменные доступны за пределами класса **Building** по причине указания модификатора доступа **public**.

Для того чтобы создать конкретный объект типа **Building**, необходимо воспользоваться следующим оператором.

Building house = new Building(); // создать объект типа Building

После выполнения этого оператора объект **house** станет экземпляром класса **Building**. Каждый объект типа **Building** будет содержать свои копии переменных экземпляра **Floors, Arena и Occupants**. Для доступа к этим переменным служит оператор доступа к члену класса, который принято называть *оператором-точкой*. Оператор-точка связывает имя объекта с именем члена класса. Ниже приведена общая форма оператора-точки.

объект.член

Например,

house.Floors = 2;

Рассмотрим пример.

class Building

{

public int Floors;

public int Area;

public int Occupants;

}

// В этом классе объявляется объект типа Building.

class BuildingDemo

{

static void Main(string[] args)

{

Building house = new Building(); // создать объект типа Building

int areaPP; // площадь на одного человека

// Присвоить значения полям в объекте house.

house.Occupants = 4;

house.Area = 2500;

house.Floors = 2;

// Вычислить площадь на одного человека.

areaPP = house.Area / house.Occupants;

Console.WriteLine("Дом имеет: {0} этажа", house.Floors);

Console.WriteLine(" {0} жильца", house.Occupants);

Console.WriteLine(" {0} кв. футов общей площади,", house.Area);

Console.WriteLine(" из них {0} приходится на одного человека", areaPP);

Console.ReadKey();

}

}

}

В данном примере:

Эта программа состоит из двух классов: Building и BuildingDemo. В последнем сначала создается экземпляр house класса Building с помощью метода Main(), а затем в коде метода Main() осуществляется доступ к переменным экземпляра house для присваивания им значений и последующего использования этих значений.

Если разделить рассматриваемую здесь программу на два файла, для ее компилирования можно воспользоваться следующей командной строкой.

csc Building.cs BuildingDemo.cs

Если пользоваться средой VS, нужно ввести оба упомянутых выше файла в свой проект и затем скомпоновать их.

Рассмотрим следующий пример.

class Building

{

public int Floors;

public int Area;

public int Occupants;

}

// В этом классе объявляются два объекта типа Building.

class BuildingDemo

{

static void Main(string[] args)

{

Building house = new Building();

Building office = new Building();

int areaPP; // площадь на одного человека

// Присвоить значения полям в объекте house.

house.Occupants = 4;

house.Area = 2500;

house.Floors = 2;

// Присвоить значения полям в объекте office.

office.Occupants = 25;

office.Area = 4200;

office.Floors = 3;

// Вычислить площадь на одного человека в жилом доме.

areaPP = house.Area / house.Occupants;

Console.WriteLine("Дом имеет: {0} этажа", house.Floors);

Console.WriteLine(" {0} жильца", house.Occupants);

Console.WriteLine(" {0} кв. футов общей площади,", house.Area);

Console.WriteLine(" из них {0} приходится на одного человека", areaPP);

// Вычислить площадь на одного человека в офисе.

areaPP = office.Area / office.Occupants;

Console.WriteLine("Офис имеет: {0} этажа", office.Floors);

Console.WriteLine(" {0} жильца", office.Occupants);

Console.WriteLine(" {0} кв. футов общей площади,", office.Area);

Console.WriteLine(" из них {0} приходится на одного человека", areaPP);

Console.ReadKey();

}

}

}

В данном примере:

Как видно, данные из объекта house полностью отделены от данных, содержащихся в объекте office.

### Создание объектов

В предыдущих примерах программ для объявления объекта типа **Building** использовалась следующая строка кода.

Building house = new Building();

Данная строка объявления выполняет три функции. Во-первых, объявляется переменная **house**, относящаяся к типу класса **Building**. Сама эта переменная не является объектом, а лишь переменной, которая может *ссылаться* на объект. Во-вторых, создается конкретная, физическая копия объекта. Это делается с помощью оператора **new**. И наконец, переменной **house** присваивается ссылка на данный объект. Таким образом, после выполнения анализируемой строки объявленная переменная **house** ссылается на объект типа **Building**.

Оператор **new** во время выполнения распределяет память для объекта и возвращает ссылку на него, которая затем сохраняется в переменной. Как и следовало ожидать, объявление переменной **house** можно отделить от создания объекта, на который она ссылается.

Building house; // объявить ссылку на объект

house = new Building(); // распределить память для объекта типа Building

То обстоятельство, что объекты классов доступны по ссылке, объясняет, почему классы называются *ссылочными типами*. Главное отличие типов значений от ссылочных типов заключается в том, что именно содержит переменная каждого из этих типов. Так, переменная типа значения содержит конкретное значение (например, **int** x = 10), в то время как переменная **house** содержит не сам объект, а лишь ссылку на него.

### Переменные ссылочного типа и присваивание

В операции присваивания переменные ссылочного типа действуют иначе, нежели переменные типа значения. Когда одна переменная типа значения присваивается другой, ситуация оказывается довольно простой: переменная, находящаяся в левой части оператора присваивания, получает копию значения переменной, находящейся в правой части этого оператора. Когда же одна переменная ссылки на объект присваивается другой, то ситуация несколько усложняется, поскольку такое присваивание приводит к тому, что переменная, находящаяся в левой части оператора присваивания, ссылается на тот же самый объект, на который ссылается переменная, находящаяся в правой части этого оператора. Сам же объект не копируется. В силу этого отличия присваивание переменных ссылочного типа может привести к несколько неожиданным результатам. Рассмотрим фрагмент кода:

Building house1 = new Building();

Building house2 = house1;

В данном случае обе переменные ссылаются на один и тот же объект. Например, после очередного присваивания

house1.Area = 2600;

оба метода **WriteLine()**

Console.WriteLine(house1.Area);

Console.WriteLine(house1.Area);

выводят одно и то же значение: 2600.

Несмотря на то что обе переменные, **house1** и **house2**, ссылаются на один и тот же объект, они никак иначе не связаны друг с другом. Например, в результате следующей последовательности операций присваивания просто изменяется объект, на который ссылается переменная **house2**.

Building house1 = new Building();

Building house2 = house1;

Building house3 = new Building();

house2 = house3; // теперь обе переменные (house2 и house3), ссылаются на один и тот же объект.

После выполнения этой последовательности операций присваивания переменная house2 ссылается на тот же самый объект, что и переменная house3. Ссылка на объект в переменной house1 не меняется.

### Методы

Переменные экземпляра и методы являются двумя основными составляющими классов. *Методы* представляют собой подпрограммы, которые манипулируют данными, определенными в классе, а во многих случаях они предоставляют доступ к этим данным.

В нашем предыдущем примере площадь на одного человека лучше всего вычислять в самом классе **Building**, просто потому, что так легче понять сам характер вычисления.

class Building

{

public int Floors;

public int Area;

public int Occupants;

// Вывести площадь на одного человека.

public void AreaPerPerson()

{

Console.WriteLine("{0} приходится на одного человека", Area/Occupants);

}

}

// Использовать метод AreaPerPerson().

class BuildingDemo

{

static void Main(string[] args)

{

Building house = new Building();

Building office = new Building();

// Присвоить значения полям в объекте house.

house.Occupants = 4;

house.Area = 2500;

house.Floors = 2;

// Присвоить значения полям в объекте office.

office.Occupants = 25;

office.Area = 4200;

office.Floors = 3;

// Вычислить площадь на одного человека в жилом доме.

Console.WriteLine("Дом имеет: {0} этажа", house.Floors);

Console.WriteLine(" {0} жильца", house.Occupants);

Console.WriteLine(" {0} кв. футов общей площади,", house.Area);

house.AreaPerPerson();

// Вычислить площадь на одного человека в офисе.

Console.WriteLine("Офис имеет: {0} этажа", office.Floors);

Console.WriteLine(" {0} жильца", office.Occupants);

Console.WriteLine(" {0} кв. футов общей площади,", office.Area);

office.AreaPerPerson();

Console.ReadKey();

}

}

}

В данном примере:

Получаем результат аналогичный предыдущему примеру.

Проанализируем следующую строку:

house.AreaPerPerson();

Вызывается метод **AreaPerPerson()** для объекта **house**. Это означает, что данный метод вызывается относительно объекта, на который ссылается переменная **house**, и для этой цели служит оператор-точка. После вызова метода **AreaPerPerson()** ему передается управление программой. По его завершении управление передается обратно вызывающей части программы, выполнение которой возобновляется со строки кода, следующей после вызова данного метода. В первом случае выводится площадь на одного человека в здании, определенном в объекте **house**. Во втором случае выводится площадь на одного человека в здании, определенном в объекте **office**. Таким образом, при каждом вызове метода **AreaPerPerson()** выводится площадь на одного человека для указанного объекта. Если в методе используется переменная экземпляра, определенная в его классе, то делается это непосредственно, без указания явной ссылки на объект и без помощи оператора-точки. Метод всегда вызывается относительно некоторого объекта его класса. Как только происходит вызов, объект становится известным. По этой причине объект не нужно указывать в методе еще раз.

В методе допускается наличие нескольких операторов **return**, особенно если имеются два или более вариантов возврата из него. Например:

public void MyMeth()

{

//...

if (done) return;

//...

if (error) return;

}

В данном примере возврат из метода происходит в двух случаях: если метод завершает свою работу или происходит ошибка.

Возвращаем величину площади вместо того, чтобы выводить ее.

class Building

{

public int Floors;

public int Area;

public int Occupants;

// Вывести площадь на одного человека.

public int AreaPerPerson()

{

return Area/Occupants;

}

}

// Использовать метод AreaPerPerson().

class BuildingDemo

{

static void Main(string[] args)

{

Building house = new Building();

Building office = new Building();

int areaPP; // площадь на одного человека

// Присвоить значения полям в объекте house.

house.Occupants = 4;

house.Area = 2500;

house.Floors = 2;

// Присвоить значения полям в объекте office.

office.Occupants = 25;

office.Area = 4200;

office.Floors = 3;

// Вычислить площадь на одного человека в жилом доме.

areaPP = house.AreaPerPerson();

Console.WriteLine("Дом имеет: {0} этажа", house.Floors);

Console.WriteLine(" {0} жильца", house.Occupants);

Console.WriteLine(" {0} кв. футов общей площади,", house.Area);

Console.WriteLine("{0} приходится на одного человека", areaPP);

// Вычислить площадь на одного человека в офисе.

areaPP = office.AreaPerPerson();

Console.WriteLine("Офис имеет: {0} этажа", office.Floors);

Console.WriteLine(" {0} жильца", office.Occupants);

Console.WriteLine(" {0} кв. футов общей площади,", office.Area);

Console.WriteLine("{0} приходится на одного человека", areaPP);

Console.ReadKey();

}

}

}

В данном примере:

Возвратили значение с помощью оператора **return**.

В приведенном ниже операторе сравниваются величины площади на одного человека для двух зданий.

if(b1.AreaPerPerson() > b2. AreaPerPerson())

Console.WriteLine("В здании b1 больше места для каждого человека");

С помощью параметризированного метода можно дополнить класс **Building** новым средством, позволяющим вычислять максимальное количество жильцов в здании, исходя из определенной величины минимальной площади на одного человека. Этим средством является метод **MaxOccupant()**.

*Конструктор* инициализирует объект при его создании. У конструктора такое же имя, как и у его класса, а с точки зрения синтаксиса он подобен методу. Однако у конструкторов нет возвращаемого типа, указываемого явно. Общая форма конструктора:

доступ имя\_класса(список\_параметров)

{

// тело конструктора

}

Как правило, конструктор используется для задания первоначальных значений переменных экземпляра, определенных в классе, или же для выполнения любых других установочных процедур, которые требуются для создания полностью сформированного объекта. Помимо этого, *доступ* обычно представляет собой модификатор доступа типа **public**, поскольку конструкторы зачастую вызываются в классе. При этом *список\_параметров* может быть как пустым, так и состоящим из одного или более указываемых параметров.

Простой пример использования конструктора.

class MyClass

{

public int x;

public MyClass()

{

x = 10;

}

class ConsDemo

{

static void Main(string[] args)

{

MyClass t1 = new MyClass();

MyClass t2 = new MyClass();

Console.WriteLine(t1.x + t2.x);

Console.ReadKey();

}

}

}

}

Выполнение данной программы дает результат 20 (t1.x + t2.x = 10 + 10 = 20). В данном примере конструктор класса **MyClass** имеет следующий вид.

public MyClass()

{

x = 10;

}

Нужно обратить внимание на то, что данный конструктор обозначается как **public**. Дело в том, что он должен вызываться из кода, определенного за пределами его класса.

Зачастую конструктор должен принимать один или несколько параметров. В конструктор параметры вводятся таким же образом, как и в метод. Для этого достаточно объявить их в скобках после имени конструктора. Пример применения параметризированного конструктора **MyClass**.

class MyClass

{

public int x;

public MyClass(int i)

{

x = i;

}

class ConsDemo

{

static void Main(string[] args)

{

MyClass t1 = new MyClass(10);

MyClass t2 = new MyClass(80);

Console.WriteLine(t1.x + t2.x);

Console.ReadKey();

}

}

}

}

Исполнение данной программы дает результат 90 (10 + 80).

В данном варианте конструктора **MyClass()** определен параметр **i**, с помощью которого инициализируется переменная экземпляра **x**. Поэтому при выполнении следующей строки кода:

MyClass t1 = new MyClass(10);

параметру **i** передается значение, которое затем присваивается переменной **x**.

Класс **Building** можно усовершенствовать, добавив в него конструктор, автоматически инициализирующий поля **Floors, Area** и **Occupants** при создании объекта. Нужно обратить особое внимание на то, как создаются объекты класса **Building**.

class Building

{

public int Floors; // количество этажей

public int Area; // общая площадь здания

public int Occupants; // количество жильцов

// Параметризированный конструктор для класса Building.

public Building(int f, int a, int o)

{

Floors = f;

Area = a;

Occupants = o;

}

// Возвратить площадь на одного человека.

public int AreaPerPerson()

{

return Area / Occupants;

}

//Возвратить максимальное количество человек, занимающих здание,

// исходя из заданной минимальной площади на одного человека.

public int MaxOccupant(int minArea)

{

return Area / minArea;

}

// Использовать параметризированный конструктор класса Building.

class BuildingDemo

{

static void Main(string[] args)

{

Building house = new Building(2, 2500, 4);

Building office = new Building(3, 4200, 25);

Console.WriteLine("Максимальное количество человек в доме,");

Console.WriteLine("если на каждого человека должно приходиться");

Console.WriteLine("300 квадратных футов: {0}", house.MaxOccupant(300));

Console.WriteLine("\n");

Console.WriteLine("Максимальное количество человек в офисе,");

Console.WriteLine("если на каждого человека должно приходиться");

Console.WriteLine("300 квадратных футов: {0}", office.MaxOccupant(300));

Console.ReadKey();

}

}

}

}

Результат выполнения этой программы оказывается таким же, как и в предыдущей ее версии.

Рассмотрим более подробно оператор **new**. В отношении классов общая форма оператора **new** такова:

new имя\_класса (список\_аргументов);

Если в классе не определен его собственный конструктор, то в операторе **new** будет использован конструктор, предоставляемый в C# по умолчанию. Следовательно, оператор **new** может быть использован для создания объекта, относящегося к классу любого типа.

Благодаря тому что основные типы данных, например, **int** или **char**, не преобразуются в ссылочные типы, существенно повышается производительность программы.

Оператор **new** разрешается использовать вместе с типами значений, как показывает следующий пример.

int i = new int();

При этом для типа **int** вызывается конструктор, инициализирующий по умолчанию переменную **i** нулевым значением. В качестве примера рассмотрим такую программу.

class newValue

{

static void Main()

{

int i = new int(); // инициализировать переменную i нулевым значением.

Console.WriteLine("Значение переменной i равно: {0}", i);

Console.ReadKey();

}

}

Результат данной программы: Значение переменной i равно 0.

Если не применить оператор **new**, переменная **i** окажется неинициализированной. В общем, обращение к оператору **new** для любого типа значения приводит к вызову конструктора, используемого по умолчанию для данного типа. Но в этом случае память не распределяется динамически. Откровенно говоря, в программировании обычно не принято пользоваться оператором **new** вместе с типами значений.

«Сборка мусора» – это освобождение части оперативной памяти от объектов, на которые отсутствуют ссылки.

В языке C# имеется возможность определить метод, который будет вызываться непосредственно перед окончательным уничтожением объекта системой «сборки мусора». Такой метод называется *деструктором* и может использоваться в ряде особых случаев, чтобы гарантировать четкое окончание срока действия объекта. Следует отметить, что деструкторы являются весьма специфическими средствами, применяемыми только в редких, особых случаях. Как правило, они не нужны.

Ниже приведена общая форма деструктора:

~имя\_класса ()

{

// код деструктора

}

Деструктор объявляется аналогично конструктору, за исключением того, что перед его именем указывается знак «тильда» (~). Нужно обратить внимание на то, что у деструктора отсутствуют возвращаемый тип и передаваемые ему аргументы.

Для того чтобы добавить деструктор в класс, достаточно включить его в качестве члена. Он вызывается всякий раз, когда предполагается утилизация объекта его класса. В деструкторе можно указать те действия, которые следует выполнить перед тем, как уничтожить объект.

class Destruct

{

public int x;

public Destruct(int i)

{

x = i;

}

// Вызывается при утилизации объекта.

~Destruct()

{

Console.WriteLine("Уничтожить {0}", x);

}

// Создать объект и тут же уничтожить его.

public void Generator(int i)

{

Destruct o = new Destruct(i);

}

class DestructDemo

{

static void Main(string[] args)

{

int count;

Destruct ob = new Destruct(0);

/\* А теперь создать большое число объектов.

В какой-то момент произойдет "сборка мусора".

Примечание: для того чтобы активизировать "сборку мусора",

возможно, придется увеличить число создаваемых объектов. \*/

for (count = 1; count < 100000; count++)

ob.Generator(count);

Console.WriteLine("Готово!");

Console.ReadKey();

}

}

}

}

Эта программа работает следующим образом. Конструктор инициализирует переменную **x** известным значением. В данном примере переменная **x** служит в качестве идентификатора объекта. Деструктор выводит значение переменной **x**, когда объект утилизируется. Особый интерес вызывает метод **Generator()**, который создает и тут же уничтожает объект типа **Destruct**. Сначала в классе **DestructDemo** создает исходный объект **ob** типа **Destruct**, а потом осуществляется поочередное создание и уничтожение 100 тыс. объектов. В разные моменты этого процесса происходит «сборка мусора». Несмотря на это, в какой-то момент начинают появляться сообщения, формируемые деструктором.

И еще одно важное замечание: метод **WriteLine()** вызывается в деструкторе **~Destruct()** исключительно ради наглядности данного примера его использования. Как правило, деструктор должен воздействовать только на переменные экземпляра, определенные в его классе.

Представим ключевое слово **this**. Когда метод вызывается, ему автоматически передается ссылка на вызывающий объект, то есть на тот объект, для которого вызывается данный метод. Эта ссылка обозначается ключевым словом **this**. Следовательно, ключевое слово **this** обозначает именно тот объект, по ссылке на который действует вызываемый метод.

Для начала рассмотрим пример программы, в которой создается класс **Rect**, инкапсулирующий ширину и высоту прямоугольника и включающий в себя метод **Area()**, возвращающий площадь прямоугольника.

class Rect

{

public int Width;

public int Height;

public Rect(int w, int h)

{

Width = w;

Height = h;

}

public int Area()

{

return Width \* Height;

}

class UseRect

{

static void Main(string[] args)

{

Rect r1 = new Rect(4, 5);

Rect r2 = new Rect(7, 9);

Console.WriteLine("Площадь прямоугольника r1: {0}", r1.Area());

Console.WriteLine("Площадь прямоугольника r2: {0}", r2.Area());

}

}

}

}

В данном случае return Width \* Height; можно записать как return this.Width \* this.Height;

В этом операторе ключевое слово **this** обозначает объект, для которого вызван метод **Area()**. Следовательно, в выражении **this.Width** делается ссылка на копию переменной **Width** данного объекта, а в выражении **this.Height** – ссылка на копию переменной **Height** этого же объекта. Так, если бы метод **Area()** был вызван для объекта **x**, то ключевое слово **this** в приведенном выше операторе обозначало бы ссылку на объект **x**. Написание оператора без ключевого слова **this** представляет собой не более чем сокращенную форму записи.

Ключевое слово **this** можно также использовать в конструкторе. В этом случае оно обозначает объект, который конструируется.

В качестве примера приведем весь класс **Rect**, написанный с использованием ссылки **this**.

class Rect

{

public int Width;

public int Height;

public Rect(int w, int h)

{

this.Width = w;

this.Height = h;

}

public int Area()

{

return this.Width \* this.Height;

}

class UseRect

{

static void Main(string[] args)

{

Rect r1 = new Rect(4, 5);

Rect r2 = new Rect(7, 9);

Console.WriteLine("Площадь прямоугольника r1: {0}", r1.Area());

Console.WriteLine("Площадь прямоугольника r2: {0}", r2.Area());

}

}

}

}

В действительности ключевое слово **this** не используется приведенным выше способом в программировании на C#, поскольку это практически ничего не дает, да и стандартная форма записи намного проще и понятнее. Тем не менее, данному ключевому слову можно найти не одно полезное применение. Например, в синтаксисе C# допускается называть параметр или локальную переменную тем же именем, что и у переменной экземпляра. В этом случае имя локальной переменной *скрывает* переменную экземпляра. Для доступа к скрытой переменной экземпляра и служит ключевое слово **this**. Например, приведенный ниже код является правильным с точки зрения синтаксиса C# способом написания конструктора **Rect()**.

public Rect(int Width, int Height)

{

this.Width = Width;

this.Height = Height;

}

В этом варианте написания конструктора **Rect()** имена параметров совпадают с именами переменных экземпляра, а следовательно, скрывают их. Но для «обнаружения» скрытых переменных служит ключевое слово **this**.

## Массивы

class Program

{

static void Main(string[] args)

{

int[] array = new int[10];

for(int i = 0; i < 10; i++)

{

array[i] = i;

}

for(int i = 0; i < 10; i++)

{

Console.WriteLine("array [{0}] = {1}", array[i], i);

}

Console.ReadKey();

}

}

}

Результат данной программы:

array[0] = 0;

array[1] = 1;

array[2] = 2;

array[3] = 3;

и т.д.

Рассмотрим пример со средним арифметическим значений элементов массива.

class Program

{

static void Main(string[] args)

{

double avg = 0;

double[] array = new double[5];

array[0] = 2.6;

array[1] = 6.4;

array[2] = 1.8;

array[3] = 7.9;

array[4] = 5.7;

for(int i = 0; i < 5; i++)

{

avg += array[i];

}

Console.WriteLine(avg / 5);

Console.ReadKey();

}

}

}

Результат данной программы: 4,88.

Аналог предыдущей программы, только с использованием блока инициализатора.

class Program

{

static void Main(string[] args)

{

double avg = 0;

double[] array = { 2.6, 6.4, 1.8, 7.9, 5.7 };

for(int i = 0; i < 5; i++)

{

avg += array[i];

}

Console.WriteLine(avg / 5);

Console.ReadKey();

}

}

}

Результат вычисления данной программы аналогичен результату предыдущей.

Несмотря на избыточность формы инициализации массива new int[] перед блоком инициализатора, она оказывается полезной в том случае, если новый массив присваивается уже существующей переменной ссылки на массив. Например:

int[] array;

array = new int[] {99, 10, 100, 18, 78};

В данном случае переменная **array** объявляется в первом операторе и инициализируется во втором.

При инициализации массива его размер можно указывать явным образом, но этот размер должен совпадать с числом инициализаторов. Еще один пример инициализации:

int[] array = new int[5] {99, 10, 100, 18, 78};

Пример двумерного массива.

{

class Program

{

static void Main(string[] args)

{

int[,] array = new int[3, 4];

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 4; j++)

{

array[i, j] = (i \* 4) + j + 1;

Console.WriteLine(array[i, j]);

}

Console.WriteLine();

}

Console.ReadKey();

}

}

}

Результат данной программы: 1234 5678 9101112. Присвоили двумерному массиву значения от 1 до 12.

Рассмотрим пример трехмерного массива, который содержит матрицу 3х3х3. После заполнения массива значения его элементов суммируются по одной из диагоналей.

class Program

{

static void Main(string[] args)

{

int[,,] array = new int[3, 3, 3];

int sum = 0;

int a = 1;

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 3; j++)

{

for(int k = 0; k < 3; k++)

{

array[i, j, k] = a++;

}

}

}

sum = array[0, 0, 0] + array[1, 1, 1] + array[2, 2, 2];

Console.WriteLine(sum);

Console.ReadKey();

}

}

}

Результат выполнения данной программы: 42.

Для инициализации многомерного массива достаточно заключить в фигурные скобки список инициализаторов каждого его размера.

тип[,] имя\_массива =

{

{abc, abc, abc, ..., abc},

{abc, abc, abc, ..., abc},

.

.

.

{abc, abc, abc, ..., abc}

};

Пример отображение значений двухмерного массива в блоке инициализатора.

class Program

{

static void Main(string[] args)

{

int[,] array =

{

{1, 2},

{2, 4},

{3, 6},

{4, 8},

{5, 10}

};

for(int i = 0; i < 5; i++)

{

for(int j = 0; j < 2; j++)

{

Console.Write(array[i, j]);

}

Console.WriteLine("\n");

}

Console.ReadKey();

}

}

}

Результат выполнения данной программы: 1 2 2 4 3 6 4 8  5 10.

Двумерный зубчатый массив объявляется следующим образом:

тип[][] имя\_массива = new тип[размер][];

В приведенном ниже фрагменте кода объявляется ступенчатый массив **jagged**. Память сначала распределяется для его первого измерения автоматически, а затем для второго измерения вручную.

int[][] jagged = new int[3][];

jagged[0] = new int[4];

jagged[1] = new int[5];

jagged[2] = new int[6];

В следующей строке кода элементу массива **jagged**, находящемуся на позиции с координатами (2,1), присваивается значение 10.

jagged[2][1] = 10;

Рассмотрим пример создания двумерного ступенчатого массива.

class Program

{

static void Main(string[] args)

{

int[][] jagged = new int[3][];

jagged[0] = new int[4];

jagged[1] = new int[5];

jagged[2] = new int[6];

// Сохранить значения в первом массиве

for (int i = 0; i < 4; i++)

jagged[0][i] = i;

// Сохранить значения во втором массиве

for (int i = 0; i < 5; i++)

jagged[1][i] = i;

// Сохранить значения в третьем массиве

for (int i = 0; i < 6; i++)

jagged[2][i] = i;

// Вывести значения из первого массива

for (int i = 0; i < 4; i++)

Console.Write(jagged[0][i]);

Console.WriteLine("\n");

// Вывести значения из второго массива

for (int i = 0; i < 5; i++)

Console.Write(jagged[1][i]);

Console.WriteLine("\n");

// Вывести значения из третьего массива

for (int i = 0; i < 6; i++)

Console.Write(jagged[2][i]);

Console.ReadKey();

}

}

}

Результат выполнения данной программы:

0123

012

01234

Ступенчатые массивы представляют собой массивы массивов, и поэтому они не обязательно должны состоять из одномерных массивов. Например, в приведенной ниже строке кода создается массив двумерных массивов.

int[][,] jagged = new int[3][,];

В следующей строке кода элементу массива **jagged[0]** присваивается ссылка на массив размерами 4х2.

jagged[0] = new int [4,2];

А в приведенной ниже строке кода элементу массива **jagged[0] [1, 0]** присваивается значение переменной **i**.

jagged[0][1,0] = i;

Присваивание значения одной переменной ссылки на массив другой переменной, по существу, означает, что обе переменные ссылаются на один и тот же массив, и в этом отношении массивы ничем не отличаются от любых других объектов. Такое присваивание не приводит ни к созданию копии массива, ни к копированию содержимого одного массива в другой. В качестве примера рассмотрим программу по присваиванию ссылок на массивы.

class Program

{

static void Main(string[] args)

{

int[] array1 = new int[10];

int[] array2 = new int[10];

int i;

for (i = 0; i < 10; i++)

{

array1[i] = i;

}

for (i = 0; i < 10; i++)

{

array2[i] = -i;

}

Console.Write("Содержимое массива array1: ");

for(i = 0; i < 10; i++)

{

Console.Write("{0} ", array1[i]);

}

Console.WriteLine("\n");

Console.Write("Содержимое массива array2: ");

for(i = 0; i < 10; i++)

{

Console.Write("{0} ", array2[i]);

}

Console.WriteLine("\n");

array2 = array1; // теперь array2 ссылается на array1

Console.Write("Содержимое массива array2 после присваивания: ");

for(i = 0; i < 10; i++)

{

Console.Write("{0} ", array2[i]);

}

Console.WriteLine("\n");

// Далее оперировать массивом array1 посредством

// переменной ссылки на массив array2.

array2[3] = 99;

Console.Write("Содержимое массива array1 после изменения \nпосредством переменной array2: ");

for(i = 0; i < 10; i++)

{

Console.Write("{0} ", array1[i]);

}

Console.ReadKey();

}

}

}

Выполнение этой программы приводит к следующему результату.

Содержимое массива array1: 0 1 2 3 4 5 6 7 8 9

Содержимое массива array2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9

Содержимое массива array2 после присваивания: 0 1 2 3 4 5 6 7 8 9

Содержимое массива array1 после изменения посредством

Переменной array2: 0 1 2 99 4 5 6 7 8 9

Как видно, после присваивания переменной array2 значения переменной array1 обе переменные ссылки на массив ссылаются на один и тот же объект.

Реализация в C# массивов в виде объектов дает целый ряд преимуществ. Одно из них заключается в том, что с каждым массивом связано свойство **Length**, содержащее число элементов, из которых может состоять массив. Следовательно, у каждого массива имеется специальное свойство, позволяющее определить его длину. Ниже приведен пример программы, в которой демонстрируется это свойство.

class Program

{

static void Main(string[] args)

{

int[] array = new int[10];

// Использовать свойство Length для инициализации массива array.

Console.Write("Длина массива array равна: {0}", array.Length);

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

{

array[i] = i \* i;

}

Console.WriteLine("\n");

// А теперь воспользоваться свойством Length

// для вывода содержимого массива array.

Console.Write("Содержимое массива array: ");

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

{

Console.Write("{0} ", array[i]);

}

Console.ReadKey();

}

}

}

При выполнении этой программы получается следующий результат.

Длина массива array равна: 10

Содержимое массива array: 0 1 4 9 16 25 36 49 64 81

Нужно иметь ввиду, что значение свойства **Length** никак не отражает число элементов, которые в нем используются на самом деле. Свойство **Length** содержит лишь число элементов, из которых может состоять массив.

Когда запрашивается длина многомерного массива, то возвращается общее число элементов, из которых может состоять массив, как в приведенном ниже примере кода.

class Program

{

static void Main(string[] args)

{

int[,,] array = new int[10, 5, 6];

Console.WriteLine("Длина массива array равна: {0}", array.Length);

Console.ReadKey();

}

}

}

Результат выполнения данной программы:

Длина массива array равна: 300

Как подтверждает приведенный выше результат, свойство **Length** содержит число элементов, из которых может состоять массив (в данном случае – 300 (10 х 5 х 6) элементов). Тем не менее, данное свойство нельзя использовать для определения длины массива в отдельном его измерении.

В качестве примера свойство **Length** используется в приведенной ниже программе с целью смены местами содержимого элементов массива, скопировав их в обратном порядке в другой массив.

class Program

{

static void Main(string[] args)

{

int i, j;

int[] array1 = new int[10];

int[] array2 = new int[10];

Console.Write("Исходное содержимое массива: ");

for(i = 0; i < array1.Length; i++)

{

array1[i] = i;

Console.Write("{0} ", array1[i]);

}

for(i = 0; i < array2.Length; i++)

{

}

Console.WriteLine("\n");

// Скопировать элементы массива array1 в массив array2 в обратном порядке.

if(array2.Length >= array1.Length) // проверить, достаточно ли длины массива array2

{

for(i = 0, j = array1.Length-1; i < array2.Length; i++, j--)

{

array2[j] = array1[i];

}

}

Console.Write("Содержимое массива в обратном порядке: ");

for(i = 0; i < array2.Length; i++)

{

Console.Write("{0} ", array2[i]);

}

Console.ReadKey();

}

}

}

Выполнение этой программы дает следующий результат:

Исходное содержимое массива: 0 1 2 3 4 5 6 7 8 9

Содержимое массива в обратном порядке: 9 8 7 6 5 4 3 2 1 0

Особый случай предоставляет применение свойства **Length** при обращении со ступенчатыми массивами. В этом случае с помощью данного свойства можно получить длину каждого массива, составляющего ступенчатый массив. В качестве примера рассмотрим программу, в которой имитируется работа центрального процессора в сети, состоящей из четырех узлов.

class Program

{

static void Main(string[] args)

{

int[][] jagged = new int[4][];

jagged[0] = new int[3];

jagged[1] = new int[7];

jagged[2] = new int[2];

jagged[3] = new int[5];

int i, j;

// Сфабриковать данные об использовании ЦП.

Console.WriteLine("Общее количество узлов сети: {0}", jagged.Length);

for (i = 0; i < jagged.Length; i++)

{

for(j = 0; j < jagged[i].Length; j++)

{

jagged[i][j] = i \* j + 70;

}

}

Console.WriteLine("\n");

for (i = 0; i < jagged.Length; i++)

{

for (j = 0; j < jagged[i].Length; j++)

{

Console.WriteLine("Использование в узле сети {0} ЦП {1}: {2}%", i, j, jagged[i][j]);

}

Console.WriteLine("\n");

}

Console.ReadKey();

}

}

}

При выполнении этой программы получается следующий результат:

Общее количество узлов сети: 4

Использование в узле 0 ЦП 0: 70%

Использование в узле 0 ЦП 1: 70%

Использование в узле 0 ЦП 2: 70%

Использование в узле 1 ЦП 0: 70%

Использование в узле 1 ЦП 1: 71%

Использование в узле 1 ЦП 2: 72%

Использование в узле 1 ЦП 3: 73%

Использование в узле 1 ЦП 4: 74%

Использование в узле 1 ЦП 5: 75%

Использование в узле 1 ЦП 6: 76%

Использование в узле 2 ЦП 0: 70%

Использование в узле 2 ЦП 1: 72%

Использование в узле 3 ЦП 0: 70%

Использование в узле 3 ЦП 1: 73%

Использование в узле 3 ЦП 2: 76%

Использование в узле 3 ЦП 3: 79%

Использование в узле 3 ЦП 4: 82%

Нужно обратить особое внимание на то, как свойство **Length** используется в ступенчатом массиве **jagged**. Напомним, что двумерный ступенчатый массив представляет собой массив массивов. Следовательно, когда используется выражение

**jagged.Length**

то в нем определяется число массивов, хранящихся в массиве **jagged** (в данном случае – четыре массива). Для получения длины любого отдельного массива, составляющего ступенчатый массив, служит следующее выражение:

**jagged[i].Length**

Рассмотрим создание неявно типизированного массива. Такой массив объявляется с помощью ключевого слова **var**, но без последующих квадратных скобок []. Кроме того, неявно типизированный массив должен быть непременно инициализирован, поскольку по типу инициализаторов определяется тип элементов данного массива. Все инициализаторы должны быть одного и того же согласованного типа. Ниже приведен пример объявления неявно типизированного массива.

**var array = new[] {1, 2, 3, 4, 5};**

В данном примере создается массив типа **int**, состоящий из пяти элементов. Ссылка на этот массив присваивается переменной **array**. Следовательно, тип этой переменной соответствует типу **int** массива, состоящего из пяти элементов. Нужно обратить внимание на то, что в левой части приведенного выше выражения отсутствуют квадратные скобки []. А в правой части этого выражения, где происходит инициализация массива, квадратные скобки присутствуют. В данном контексте они обязательны.

Рассмотрим еще один пример, в котором создается двумерный массив типа **double**.

**var array = new[ , ] { {1.1, 2.2}, {3.3, 4.4}, {5.5, 6.6} };**

В данном случае получается массив **array** размерами 2х3.

Объявлять можно также неявно типизированные ступенчатые массивы. В качестве примера рассмотрим следующую программу.

class Program

{

static void Main(string[] args)

{

var jagged = new[]

{

new[] {1, 2, 3, 4},

new[] {9, 8, 7},

new[] {11, 12, 13, 14, 15},

};

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

{

for(int j = 0; j < jagged[i].Length; j++)

{

Console.Write("{0} ", jagged[i][j]);

}

Console.WriteLine();

}

Console.ReadKey();

}

}

}

Результат выполнения данной программы:

1 2 3 4

9 8 7

11 12 13 14 15

Как видно, оператор **new[]** используется в этом объявлении двояким образом. Во-первых, этот оператор создает массив массивов. И во-вторых, он создает каждый массив в отдельности, исходя из количества инициализаторов и их типа. Как и следовало ожидать, все инициализаторы отдельных массивов должны быть одного и того же типа. Таким образом, к объявлению любого неявно типизированного ступенчатого массива применяется тот же самый общий подход, что и к объявлению обычных ступенчатых массивов. Неявно типизированные массивы чаще всего применяются в LINQ-ориентированных запросах.

## Оператор цикла foreach

Оператор **foreach** служит для циклического обращения к элементам *коллекции*, представляющей собой группу объектов. В C# определено несколько видов коллекций, каждая из которых является массивом. Ниже приведена общая форма оператора цикла **foreach**.

foreach(тип имя\_переменной\_цикла in коллекция) оператор;

Здесь *тип имя\_переменной\_цикла* обозначает тип и имя переменной управления циклом, которая получает значение следующего элемента коллекции на каждом шаге выполнения цикла **foreach**. А *коллекция* обозначает циклически опрашиваемую коллекцию, которая здесь и далее представляет собой массив. Следовательно, *тип* переменной цикла должен соответствовать типу элемента массива. Кроме того, *тип* может обозначаться ключевым словом **var**. В этом случае компилятор определяет тип переменной цикла, исходя из типа элемента массива. Это может оказаться полезным для работы с определенного рода запросами. Но, как правило, тип указывается явным образом.

Оператор цикла **foreach** действует следующим образом. Когда цикл начинается, первый элемент массива выбирается и присваивается переменной цикла. На каждом последующем шаге итерации выбирается следующий элемент массива, который сохраняется в переменной цикла. Цикл завершается, когда все элементы массива окажутся выбранными. Следовательно, оператор **foreach** циклически опрашивает массив по отдельным его элементам от начала и до конца.

Следует, однако, иметь ввиду, что переменная цикла в операторе **foreach** служит только для чтения. Это означает, что присваивая этой переменной новое значение, нельзя изменить содержимое массива.

Ниже приведен простой пример использования оператора цикла **foreach**. В этом примере сначала создается целочисленный массив и задается ряд его первоначальных значений, а затем эти значения выводятся, а по ходу дела вычисляется их сумма.

class Program

{

static void Main(string[] args)

{

int sum = 0;

int[] array = new int[10];

// Задать первоначальные значения элементов массива array.

for(int i = 0; i < 10; i++)

{

array[i] = i;

}

// Использовать цикл foreach для вывода значений

// элементов массива и подсчитать их суммы.

foreach(int x in array)

{

Console.WriteLine("Значение элемента равно: {0}", x);

sum += x;

}

Console.WriteLine("Сумма равна: {0}", sum);

Console.ReadKey();

}

}

}

Выполнение программы дает следующий результат:

Значение элемента равно: 0

Значение элемента равно: 1

Значение элемента равно: 2

Значение элемента равно: 3

Значение элемента равно: 4

Значение элемента равно: 5

Значение элемента равно: 6

Значение элемента равно: 7

Значение элемента равно: 8

Значение элемента равно: 9

Сумма равна: 45

Как видно, оператор **foreach** циклически опрашивает массив по порядку индексирования от самого первого до самого последнего его элемента.

Несмотря на то что цикл **foreach** повторяется до тех пор, пока не будут опрошены все элементы массива, его можно завершить преждевременно, воспользовавшись оператором **break**. Ниже приведен пример программы, в которой суммируются только пять первых элементов массива **array**.

class Program

{

static void Main(string[] args)

{

int sum = 0;

int[] array = new int[10];

for(int i = 0; i < 10; i++)

{

array[i] = i;

}

foreach(int x in array)

{

Console.WriteLine("Значение элемента равно: {0}", x);

sum += x;

if(x == 4)

{

break; // прервать цикл, как только индекс массива достигнет 4.

}

}

Console.WriteLine("Сумма первых 5 элементов равна: {0}", sum);

Console.ReadKey();

}

}

}

Результат выполнения данной программы:

Значение элемента равно: 0

Значение элемента равно: 1

Значение элемента равно: 2

Значение элемента равно: 3

Значение элемента равно: 4

Сумма первых 5 элементов равна: 10

Совершенно очевидно, что цикл **foreach** завершается после выбора и вывода значения пятого элемента массива.

Оператор цикла **foreach** можно также использовать для циклического обращения к элементам многомерного массива. В этом случае элементы многомерного массива возвращаются по порядку следования строк от первой до последней, как демонстрирует приведенный ниже пример.

class Program

{

static void Main(string[] args)

{

int sum = 0;

int[,] array = new int[3, 5];

for(int i = 0; i < 3; i++)

{

for(int j = 0; j < 5; j++)

{

array[i, j] = (i + 1) \* (j + 1);

}

}

foreach(int x in array)

{

Console.WriteLine("Значение элемента равно: {0}", x);

sum += x;

}

Console.WriteLine("Сумма равна: {0}", sum);

Console.ReadKey();

}

}

}

Результат выполнения этой программы:

Значение элемента равно: 1

Значение элемента равно: 2

Значение элемента равно: 3

Значение элемента равно: 4

Значение элемента равно: 5

Значение элемента равно: 2

Значение элемента равно: 4

Значение элемента равно: 6

Значение элемента равно: 8

Значение элемента равно: 10

Значение элемента равно: 3

Значение элемента равно: 6

Значение элемента равно: 9

Значение элемента равно: 12

Значение элемента равно: 15

Сумма равна: 90

Оператор **foreach** допускает циклическое обращение к массиву только в определенном порядке: от начала и до конца массива, поэтому его применение кажется, на первый взгляд, ограниченным. Но на самом деле это не так. В большом числе алгоритмов, самым распространенным из которых является алгоритм поиска, требуется именно такой механизм. В качестве примера ниже приведена программа, в которой цикл **foreach** используется для поиска в массиве определенного значения. Как только это значение будет найдено, цикл прервется.

class Program

{

static void Main(string[] args)

{

int[] array = new int[10];

int val = 5;

bool found = false;

for(int i = 0; i < 10; i++)

{

array[i] = i;

}

foreach(int x in array)

{

if(x == val)

{

found = true;

break;

}

}

if(found)

{

Console.WriteLine("Значение найдено!");

}

Console.ReadKey();

}

}

}

Результат выполнения данной программы: Значение найдено!

Оператор цикла **foreach** отлично подходит для такого применения, поскольку при поиске в массиве приходится анализировать каждый его элемент. К другим примерам применения оператора цикла **foreach** относится вычисление среднего, поиск минимального или максимального значения среди ряда заданных значений, обнаружение дубликатов и т.д. Оператор цикла **foreach** оказывается особенно полезным для работы с разными типами коллекций.

Рассмотрим программу нахождения минимального и максимального значения массива.

class Program

{

static void Main(string[] args)

{

int[] array = new int[10];

array[0] = 18;

array[1] = 10;

array[2] = 20;

array[3] = 3;

array[4] = 1;

array[5] = 8;

array[6] = 16;

array[7] = 13;

array[8] = 5;

array[9] = 2;

int min = array[0];

int max = array[0];

foreach (int x in array)

{

if (x < min)

{

min = x;

}

else if (x > max)

{

max = x;

}

}

Console.WriteLine("Число {0} является минимальным значением массива", min);

Console.WriteLine("Число {0} является максимальным значением массива", max);

Console.ReadKey();

}

}

}

Результат выполнения данной программы:

Число 1 является минимальным значением массива

Число 20 является максимальным значением массива

Рассмотрим пример нахождения значения массива по среднему индексу

class Program

{

static void Main(string[] args)

{

int[] array = {8, 10, 15, 16, 19, 23, 25, 6, 9, 11};

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

{

if(i == (array.Length/2))

{

Console.WriteLine("Средний индекс {0} массива содержит значение {1}", array.Length/2, array[i]);

break;

}

}

Console.ReadKey();

}

}

}

Результат выполнения данной программы:

Средний индекс 5 массива содержит значение 23

Рассмотрим пример пузырьковой сортировки массива.

class Program

{

static void Main(string[] args)

{

int[] array = {5, 7, 2, 9, 4, 3, 6, 1, 8, 0};

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

{

for(int j = 0; j < array.Length - 1; j++)

{

if(array[j + 1] < array[j])

{

int temp = array[j];

array[j] = array[j + 1];

array[j + 1] = temp;

}

}

}

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

{

Console.Write(" {0}", array[i]);

}

Console.ReadKey();

}

}

}

Результат выполнения данной программы: 0 1 2 3 4 5 6 7 8 9

Рассмотрим пример быстрой сортировки массива.

class Program

{

static void Sort(int[] array, int l, int r)

{

int i = l;

int j = r;

int x = array[(l + r) / 2];

while (i <= j)

{

while (array[i] < x) i++;

while (array[j] > x) j--;

{

if (i <= j)

{

int temp = array[i];

array[i] = array[j];

array[j] = temp;

i++;

j--;

}

}

}

if(l<j)

Sort(array, l, j);

if(i<r)

Sort(array, i, r);

}

static void Main(string[] args)

{

int[] array = {35, 18, 40, 5, 15, 38, 2, 9, 22, 3};

foreach(int a in array)

{

Console.Write("{0} ", a);

}

Console.WriteLine();

Console.WriteLine("После сортировки: ");

Sort(array, 0, array.Length - 1);

foreach(int a in array)

{

Console.Write("{0} ", a);

}

Console.ReadKey();

}

}

}

Результат выполнения данной программы: 2 3 5 9 15 18 22 35 38 40

Рассмотрим пример двоичного поиска элемента в отсортированном массиве.

class Program

{

static int BinarySearch(int[] array, int k)

{

int l = 0;

int r = array.Length-1;

while (l <= r)

{

int m = (l + r) / 2;

if ((array[m] > k))

{

r = m - 1;

}

else if (array[m] < k)

{

l = m + 1;

}

else return m;

}

return -1;

}

static void Main(string[] args)

{

int[] array = { 18, 31, 44, 59, 68, 72, 81, 90, 104, 111 };

Console.WriteLine(BinarySearch(array, 90));

Console.ReadKey();

}

}

}

Результат выполнения данной программы: 7 (значение находится в элементе 7 массива).