ГЛАВА 2

Введение в типы данных и операторы

Ключевые навыки, понятия и концепции

- Основные типы C#
- Форматный вывод
- Литералы
- Инициализация переменных
- Правила области видимости (контекста) метода
- Преобразование типов и приведение
- Арифметические операторы
- Операторы отношений и логические
- Оператор присваивания
- Выражения

В основе любого языка программирования лежат его типы данных и операторы, и C# не исключение. Эти элементы определяют пределы применимости языка и определяют виды задач, к которым он может быть применен. Как и следовало ожидать, C# поддерживает богатый выбор типов данных и операторов, что делает его подходящим для программирования широкого диапазона задач.

Типы данных и операторы — большая тема. Мы начнем здесь с изучения основополагающих типов данных и обычных операторов в C#. Мы также более подробно исследуем переменные и выражения.

Почему типы данных так важны

Типы данных особенно важны в C#, потому что это язык со строгим контролем типов. Это означает, что во всех операциях компилятор следит за совместимостью типов. Незаконные операции не откомпилируются. Таким образом, строгая проверка контроля соответствия типов предотвращает ошибки и повышает надежность. Чтобы осуществить строгий контроль соответствия типов, все переменные, выражения и значения имеют тип. Например, нет понятия (концепции) переменной “без типа”. Кроме того, тип значения определяет, какие операции допускаются на нем. Операцию, допустимую на одном типе, нельзя выполнить на другом.

Типы значений в C#

C# содержит две общие конструкции встроенных типов данных — *типы значений* и *ссылочные типы*. Различие между двумя типами состоит в том, что содержит переменная. В случае типа значения переменная содержит фактическое значение, такое как 101 или 98,6. В случае ссылочного типа переменная содержит ссылку на значение.

Обычно используемый ссылочный тип — класс, но обсуждение классов и ссылочных типов мы отложим на более поздний срок. Типы значения описаны здесь.

В ядре C# 13 типов значений, приведенных в табл. 2.1. Все вместе, они упоминаются как *простые типы*. Их называют простыми типами, потому что они состоят из единственного (отдельного) значения. (Другими словами, они не являются составными объектами из двух или больше значений.) Они формируют основу системы типов в C#, обеспечивая основные элементы данных нижнего уровня, которые обрабатывает программа. Простые типы называются также иногда *примитивными типами*.

C# строго определяет диапазон и поведение каждого простого типа. Из-за требований мобильности и поддержки многоязыкового программирования C# бескомпромиссен в этом отношении. Например, тип `int` тот же самый во всех средах выполнения. Нет никакой потребности переписывать код, чтобы он соответствовал определенной платформе. Строгое определение размера простых ти-

пов может вызвать малую потерю производительности в небольшом количестве сред, но оно необходимо, чтобы достигнуть мобильности.

ПРИМЕЧАНИЕ

В дополнение к простым типам, в C# определены три других категорий типов значения. Это перечисления, структуры и типы с неопределенными значениями, все они описаны позже в этой книге.

Целые числа

В C# определено девять типов целых чисел: `char` (символ), `byte` (байт), `sbyte`, `short` (короткий), `ushort`, `int`, `uint`, `long` и `ulong`.

Однако тип символа `char` прежде всего используется для того, чтобы представить символы, это обсуждается позже в этой главе. Остальные восемь целочисленных типов используются для числовых вычислений. Их ширина в битах и диапазоны приведены ниже.

Тип	Ширина в битах	Диапазон
`byte` (байт)	8	От 0 до 255
`sbyte`	8	От –128 до 127
`short`	16	От –32 768 до 32 767
`ushort`	16	От 0 до 65 535
`int`	32	От –2 147 483 648 до 2 147 483 647
`uint`	32	От 0 до 4 294 967 295
`long`	64	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
`ulong`	64	От 0 до 18 446 744 073 709 551 615

Таблица 2.1. Простые типы C#

Тип	Значение
`bool`	Представляет истинные или ложные значения
`byte` (байт)	Целое число без знака длиной 8 битов
`char` (символ)	Символ
`decimal` (десятичное число)	Числовой тип для финансовых вычислений
`double`	С двойной точностью с плавающей запятой
`float` (с плавающей точкой)	С одинарной точностью с плавающей запятой
`int`	Целое число
`long`	Длинное целое число
`sbyte`	Целое число со знаком длиной 8 битов
`short`	Короткое целое число
`uint`	Целое число без знака
`ulong`	Длинное целое число без знака
`ushort`	Короткое целое число без знака

Как показывает таблица, в C# определены различные типы целого числа — есть версии без знака и со знаком. Различие между и целыми числами без знака и целыми числами со знаком состоит в способе, которым интерпретируется старший бит целого числа. Если определено целое число со знаком, компилятор C# генерирует код, который предполагает, что старший бит целого числа должен использоваться как *флажок* (*признак*) *знака*. Если флажок знака 0, число неотрицательное; если это 1, число отрицательное. Отрицательные числа почти всегда представляются в *двоичном дополнительном* коде. Чтобы получить такое представление, надо все биты в числе изменить (инвертировать), а затем добавить 1 к полученному двоичному представлению числа.

Целые числа со знаком важны для очень многих алгоритмов, но диапазон их абсолютной величины составляет только половину диапазона абсолютной величины их родственников без знака. Например, число `short` 32 767 имеет представление:

```
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Для значения со знаком, если бы старший бит был равен 1, тогда число интерпретировалось бы как –1 (в предположении двоичного дополнительного формата). Однако если объявить, что это `ushort`, то если старший бит положить равным 1, число будет рассматриваться как 65 535.

Вероятно, чаще всего используется целочисленный тип `int`. Переменные типа `int` часто используются для управления циклами, индексирования массивов и для универсальной целочисленной математики. Когда нужно целое число, которое имеет диапазон больше, чем `int`, есть много вариантов. Если значение, которое нужно сохранить, без знака, можно использовать `uint`. Для больших значений со знаком, используйте `long`. Для еще больших значений без знака используйте `ulong`.

Приведенная ниже программа вычисляет число кубических дюймов, содержащихся в кубе, каждая сторона которого равна 1 миле. Поскольку это значение является очень большим, программа использует переменную типа `long`, чтобы хранить его.

```
// Вычислить число кубических дюймов в 1 кубической миле.

using System;

class Inches {

    static void Main()     {

        // Объявим две переменные типа long.
        long ci;
        long im;

        im = 5280 * 12;

        ci = im * im * im;
```

```
        Console.WriteLine("There are " + ci +
                          " cubic inches in cubic mile.");
    }
}
```

Вот вывод из программы:

```
There are 254358061056000 cubic inches in cubic mile.
```

Очевидно, результат не поместился бы в переменной типа `int` или `uint`.

Наименьшие целочисленные типы — байт `byte` и `sbyte`. Тип байта `byte` — значение без знака между 0 и 255. Переменные типа `byte` (байт) особенно полезны при работе с необработанными двоичными данными, таких как поток байтов, получаемый от некоторого устройства. Для маленьких целых чисел со знаком, используйте `sbyte`. Ниже приведен пример, в котором переменная типа `byte` используется для управления циклом `for`, который проводит суммирование 100 чисел:

```
// Используется byte (байт).

using System;

class Use_byte {
    static void Main() {
        byte x;
        int sum;

        sum = 0;
        // Используется переменная типа byte,

        // чтобы управлять циклом for.
        for (x = 1; x <= 100; x++)
            sum = sum + x;
        Console.WriteLine("Summation of 100 is " + sum);
    }
}
```

Вывод программы приведен ниже:

```
Summation of 100 is 5050
```

Поскольку цикл `for` выполняется для чисел только от 0 до 100, которые хорошо укладываются в пределах диапазона байта (тип `byte`), нет никакой потребности использовать переменную большего типа для управления циклом. Конечно, байт (тип `byte`) нельзя использовать для хранения результата суммирования, потому что 5050 лежит далеко вне этого диапазона. Именно поэтому сумма `sum` имеет тип `int`.

Если целое число не умещается в байт (т.е. в тип `byte` или `sbyte`), но меньше, чем `int` или `uint`, используйте `short` или `ushort`.

Типы с плавающей точкой

Как объяснялось в главе 1 “Основные принципы C#”, типы с плавающей точкой могут представить числа, которые имеют дробные части. Есть два вида типов с плавающей точкой, `float` (с плавающей точкой) и `double`, которые представляют числа с одинарной и с двойной точностью соответственно. Тип с плавающей точкой `float` шириной 32 бита и имеет диапазон от 1,5E–45 до 3,4E+38. Тип `double` — 64 бита шириной и имеет диапазон от 5E–324 до 1,7E+308.

Из этих двух типов обычно используется `double`. Причина этого состоит в том, что многие математические функции в библиотеке классов C# (это библиотека .NET Framework) используют значения типа `double`. Например, метод `Sqrt( )` (определен в классе `System.Math`) возвращает значение типа `double`, которое является квадратным корнем его параметра типа `double`. Здесь используется `Sqrt( )`, чтобы вычислить длину гипотенузы по заданным длинам двух катетов:

```
/*
  Используем теорему Пифагора, чтобы найти длину гипотенузы,
  если заданы длины двух катетов.
*/

using System;

class Hypot {
    static void Main() {
        double x, y, z;
        x = 3;
        y = 4;

        // Обратите внимание, как вызывается Sqrt( ).
        // Его имени предшествует название (имя) класса,
        // членом которого он является.
        z = Math.Sqrt(x * x + y * y);

        Console.WriteLine("Hypotenuse is " + z);
    }
}
```

Распечатка, полученная после прогона программы, приведена ниже:

```
Hypotenuse is 5
```

Вот еще один важный момент в предыдущем примере. Как уже упоминалось, `Sqrt( )` — член класса `Math`. Обратите внимание, как вызывается `Sqrt( )`; ему предшествует имя `Math`. Это подобно тому, как `Console` предшествует `WriteLine( )`. Правда, не все стандартные методы вызывают, сначала указывая имя их класса.

Десятичный тип decimal

Возможно наиболее интересным числовым типом в C# является десятичный `decimal`, который предназначен для использования в денежно-кредитных вычислениях. Десятичный тип `decimal` занимает 128 битов для представления значения в диапазоне от 1E–28 до 7,9E+28. Как известно, обычная арифметика с плавающей точкой подчинена разнообразию ошибок округления, когда она применяется к десятичным значениям. Десятичный тип устраняет такие ошибки и может точно представить до 28 десятичных разрядов (или 29 разрядов в некоторых случаях). Эта способность представлять десятичные значения без ошибок округления делает данный тип особенно полезным для вычислений, в которых используются суммы денег.

Ниже приведена программа, в которой десятичное число (тип `decimal`) используется в финансовом вычислении. Программа вычисляет баланс после применения нормы.

```
// Используем десятичное число (тип decimal) в финансовом вычислении.

using System;

class UseDecimal {
    static void Main() {
        decimal balance;
        decimal rate;

        // Вычислить новый баланс.
        // После десятичных значений должно стоять m или M.
        balance = 1000.10m;
        rate = 0.1m;
        balance = balance * rate + balance;

        Console.WriteLine("New balance: $" + balance);
    }
}
```

Распечатка, полученная после прогона этой программы, приведена ниже:

```
New balance: $1100.110
```

Спросите у эксперта

ВОПРОС. Другие машинные языки, с которыми я работал, не имеют десятичного типа данных `decimal`. Уникален ли в этом отношении C#?

ОТВЕТ. Десятичный тип не поддерживается в C, C++ и Java как встроенный тип. Таким образом, в пределах его прямой линии наследования он уникален.

Обратите внимание, что в программе после десятичных констант следует `m` или `M` в качестве суффикса. Это необходимо, потому что без суффикса эти значения интерпретировались бы как стандартные константы с плавающей запятой, которые не совместимы с десятичным типом данных `decimal`. (Позже в этой главе мы более подробно рассмотрим определение числовых констант.)

Символы

В C# символы — это не 8-битовые числа, как во многих других машинных языках, таких как C++. Вместо этого в C# используется Unicode. *Unicode* определяет набор символов, который может представить все символы, имеющиеся во всех земных языках. Таким образом, в C# тип символа `char` — это 16-разрядный тип без знака, имеющий диапазон от 0 до 65 535. Стандартный набор 8-битовых символов ASCII — подмножество Unicode и его коды занимают диапазон от 0 до 127. Таким образом, символы ASCII являются допустимыми символами и в C#.

Символьной переменной может быть присвоено значение, включая и символ в одинарных кавычках. Например, приведенное ниже выражение присваивает *X* переменной `ch`:

```
char ch;
ch = 'X';
```

Вы можете вывести значение символа, используя инструкцию `WriteLine( )`. Например, следующая строка выводит значение, хранимое в `ch`:

```
Console.WriteLine("This is ch: " + ch);
```

Хотя в C# символ `char` определен как целочисленный тип, его нельзя свободно смешивать с целыми числами во всех случаях. Так происходит потому, что нет никакого автоматического преобразования типов из целого числа в символ `char`. Например, следующий фрагмент недопустим:

```
char ch;
ch = 10; // ошибка, не будет работать
```

Спросите у эксперта

ВОПРОС. Почему C# использует Unicode?

ОТВЕТ. C# был разработан так, чтобы программы были пригодны для использования во всем мире. Таким образом, он должен использовать набор символов, который может представить все всемирные языки. Unicode — стандартный набор символов, разработанный явно для этой цели. Конечно, Unicode неэффективен для таких языков, как английский, немецкий, испанский и французский языки, символы которых могут содержаться в 8 битах. Но это — цена глобальной мобильности.

Причина, по которой предыдущий код не будет работать, состоит в том, что 10 — целочисленное значение и оно не будет автоматически преобразовано в символ. Таким образом, присваивание содержит несовместимые типы. Если попытаться скомпилировать этот код, появится сообщение об ошибках. Позже в этой главе мы научимся обходить это ограничение.

Тип `bool`

Тип `bool` представляет значения истина или ложь. В C# определены значения `true` (истина) и `false` (ложь) и используются зарезервированные слова `true` и `false`. Таким образом, переменная или выражение типа `bool` будут иметь одно из этих двух значений. Кроме того, нет никакого определенного преобразования между `bool` и целочисленными значениями. Например, 1 не преобразуется в `true`, а 0 не преобразуется в `false`.

Ниже приведена программа, которая демонстрирует тип `bool`:

```
// Демонстрируем значения типа bool.

using System;

class BoolDemo {
    static void Main() {
        bool b;

        b = false;
        Console.WriteLine("b is " + b);
        b = true;
        Console.WriteLine("b is now " + b);

        // Значение bool может управлять условным оператором.
        if (b) Console.WriteLine("This is executed.");
        // Отдельное значение типа bool может управлять
        // условным оператором!!!

        b = false;
        if (b) Console.WriteLine("This is not executed.");

        // Результат оператора отношения - значение типа bool.
        Console.WriteLine("88 > 17 is " + (88 > 17));
    }
}
```

Вывод, сгенерированный этой программой, приведен ниже:

```
b is False
b is now True
This is executed.
88 > 17 is True
```

В этой программе есть три интересных момента, на которые следует обратить внимание. Во-первых, когда значение `bool` выводится инструкцией

`WriteLine( )`, отображается “True” или “False”. Во-вторых, отдельного значения переменной типа `bool` достаточно, чтобы управлять условным оператором. Нет никакой потребности писать условный оператор так:

```
if(b == true) ...
```

В-третьих, результат оператора отношения, такого как <, является значением типа `bool`. Именно поэтому выражение `88 > 17` отображает значение “True”. Далее, дополнительный набор круглых скобок вокруг `88 > 17` необходим, потому что оператор + имеет более высокое старшинство, чем оператор >.

Некоторые опции вывода

Перед продолжением изучения типов данных и операторов будет полезно маленькое отклонение. До этого момента в списках выводимых данных каждая часть списка отделялась знаком “плюс”, как показано ниже:

```
Console.WriteLine("You ordered " + 2 + " items at $" + 3 + " each.");
```

Хотя это очень удобно, этот способ вывода числовой информации не позволяет управлять отображением этой информации. Например, для значения с плавающей точкой нельзя управлять количеством отображаемых десятичных знаков. Рассмотрим следующую инструкцию:

```
Console.WriteLine("Here is 10/3: " + 10.0/3.0);
```

Эта инструкция генерирует следующий вывод:

```
Here is 10/3: 3.33333333333333
```

Хотя это прекрасно для некоторых целей, отображение такого большого количества десятичных знаков не подходит для других. Например, в финансовых вычислениях обычно достаточно отобразить два десятичных знака.

Чтобы управлять форматированием числовых данных, необходимо использовать вторую форму `WriteLine( )`, показанную ниже. Эта форма позволяет внедрять информацию форматирования:

`WriteLine("`*строка формата*`", arg0, arg1, ... , argN)`

В этой версии параметры `WriteLine( )` отделены запятыми, а не знаками “плюс”. *Строка формата* содержит два элемента — обычный, с печатными символами, которые отображаются как есть, и спецификаторы формата. Спецификаторы формата имеют следующую общую форму:

{*argnum*`,` *ширина*`:` *формат*}

Здесь *argnum* указывает номер отображаемого параметра (начинающийся с нуля). Минимальная ширина поля определена *шириной*, а формат определен *форматом*.

Во время выполнения, когда спецификатор формата встречается в строке формата, соответствующий параметр, определенный *argnum*, заменяет

его и отображается. Таким образом, именно позиция спецификации формата в строке формата определяет, где будут отображены соответствующие ей данные. И ширина, и формат являются дополнительными, необязательными. Таким образом, в его самой простой форме спецификатор формата просто указывает, какой параметр нужно отобразить. Например, `{0}` указывает, что нужно отобразить *arg0*, `{1}` указывает на *arg1* и т.д.

Давайте начнем с простого примера. Инструкция

```
Console.WriteLine("February has {0} or {1} days.", 28, 29);
```

генерирует следующий вывод:

```
February has 28 or 29 days.
```

Как вы можете видеть, значением 28 заменяется `{0}`, а 29 заменяет `{1}`. Таким образом, спецификаторы формата идентифицируют местоположение последующих параметров, в этом случае 28 и 29 — отображены в строке. Кроме того, обратите внимание, что дополнительные значения отделены запятыми, а не знаками "плюс".

Вот изменение предыдущей инструкции, в которой указаны минимальные ширины поля:

```
Console.WriteLine("February has {0,10} or {1,5} days.", 28, 29);
```

Это генерирует следующий вывод:

```
February has    28 or    29 days.
```

Как вы можете видеть, добавились пробелы, чтобы заполнить неиспользованные части полей. Помните, минимальная ширина поля — это только минимальная ширина. Вывод может превысить эту ширину, если необходимо.

В предыдущих примерах к значениям непосредственно никакое форматирование не применялось. Конечно, значение спецификаторов формата состоит именно в том, что они управляют способом отображения данных. Чаще всего форматируются данные, имеющие типы десятичных значений с плавающей точкой. Один из самых простых способов определить формат состоит в том, чтобы указать шаблон для `WriteLine( )`. Чтобы сделать это, укажите пример нужного формата, используя знаки # для того, чтобы отметить цифровые позиции. Например, ниже показан лучший способ отобразить частное от 10, разделенного на 3:

```
Console.WriteLine("Here is 10/3: {0:#.##}", 10.0/3.0);
```

Вывод из этой инструкции приведен ниже:

```
Here is 10/3: 3.33
```

В этом примере шаблон — `#.##`, который указывает, что инструкция `WriteLine( )` должна отображать два десятичных знака. Важно понять, однако, что в случае необходимости `WriteLine( )` отобразит больше, чем одну цифру, слева от десятичной точки, чтобы не исказить значение.

Если нужно отобразить денежно-кредитные значения, используйте спецификатор формата C. Например,

```
decimal balance;

balance = 12323.09m;
Console.WriteLine("Current balance is {0:C}", balance);
```

Вывод из этой последовательности приведен ниже (в формате долларов США):

```
Current balance is $12,323.09
```

Попробуйте вот это. Говорите с Марсом

В его самой близкой к Земле точке своей орбиты Марс находится приблизительно на расстоянии 34 миллиона миль. Предположим, кто-то на Марсе хочет поговорить с вами. Вас интересует, велика ли задержка во времени; иными словами, за какое время радиосигнал с Земли достигает Марса? Эта программа находит ответ. Вспомните, что скорость света приблизительно равна 186 000 миль в секунду. Таким образом, чтобы вычислить задержку, нужно разделить расстояние на скорость света. Выразите задержку в секундах и минутах.

Шаг за шагом

1. Создайте новый файл по имени `Mars.cs`.
2. Чтобы вычислить задержку, используйте значения с плавающей точкой. Почему? Поскольку интервал времени будет иметь дробную часть. Вот переменные, используемые программой:

   ```
   double distance;
   double lightspeed;
   double delay;
   double delay_in_min;
   ```

3. Присвойте расстоянию `distance` и `lightspeed` следующие значения:

   ```
   distance = 34000000; // расстояние 34 000 000 миль
   lightspeed = 186000; // 186 000 миль в секунду
   ```

4. Чтобы вычислять задержку `delay`, разделите расстояние `distance` на `lightspeed`. Это позволяет мгновенно вычислить задержку `delay`.
5. Присвойте это значение `delay` и отобразите результаты. Эти шаги приведены ниже:

   ```
   delay = distance / lightspeed;
   Console.WriteLine("Time delay when talking to Mars: " +
                     delay + " seconds.");
   ```

6. Разделите количество секунд на 60, чтобы получить задержку в минутах; отобразите полученный результат с помощью следующих строк программы:

```
delay_in_min = delay / 60;
Console.WriteLine("This is " + delay_in_min + " minutes.");
```

7. Вот полная распечатка программы `Mars.cs`:

```
// Разговор с Марсом

using System;

class Mars {
    static void Main() {
        double distance;
        double lightspeed;
        double delay;
        double delay_in_min;

        distance = 34000000; // расстояние 34 000 000 миль
        lightspeed = 186000; // 186 000 миль в секунду

        delay = distance / lightspeed;

        Console.WriteLine("Time delay when talking to Mars: " +
                          delay + " seconds.");

        delay_in_min = delay / 60;

        Console.WriteLine("This is " + delay_in_min +
                          " minutes.");
    }
}
```

8. Скомпилируйте и выполните программу. Будет отображен следующий результат:

```
Time delay when talking to Mars: 182.795698924731 seconds.
This is 3.04659498207885 minutes.
```

9. Для большинства людей программа отображает слишком много десятичных мест. Чтобы улучшить удобочитаемость программы, замените в программе следующим `WriteLine( )` те инструкции, которые отображают результаты:

```
Console.WriteLine("Time delay when talking to Mars: {0:#.###} seconds",
                  delay);
Console.WriteLine("This is about {0:#.###} minutes", delay_in_min);
```

10. Перетранслируйте и выполните программу. Когда вы это сделаете, увидите следующий вывод:

```
Time delay when talking to Mars: 182.796 seconds
This is about 3.047 minutes
```

Теперь отображаются только три десятичных места.

Литералы

В C# *литералами* называются заданные значения, которые представлены в их удобочитаемой форме. Например, число 100 — это литерал. В большинстве случаев понятие литералов и работа с ними настолько интуитивны, что они использовались в той или иной форме всеми предыдущими типовыми программами. Теперь пришло время объяснить их формально.

Литерал в C# может иметь любой из типов значений. Способ представления каждого литерала зависит от его типа. Как уже указывалось, литерал-символ заключается в одинарные кавычки. Например 'a' и '%' — два символа-литерала.

Литерал в виде целого числа указывается как число без дробной части. Например, 10 и –100 — целочисленные литералы. Литералы с плавающей точкой требуют использования десятичной точки, за которой следует дробная часть числа. Например, 11,123 есть литерал с плавающей точкой. C# также позволяет также использовать экспоненциальный формат для чисел с плавающей запятой.

Поскольку C# — язык со строгим контролем типов, литералы также имеют тип. Естественно, это поднимает следующий вопрос: каков тип числовых литералов? Например, что является типом 12, 123987 или 0,23? К счастью, в C# предусмотрены некоторые понятные правила, позволяющие ответить на эти вопросы.

Во-первых, в случае целочисленного литерала тип литерала — это наименьший целочисленный тип, который может хранить его, начиная с `int`. Таким образом, целочисленный литерал имеет тип `int`, `uint`, `long` или `ulong` в зависимости от его значения. Во-вторых, литералы с плавающей точкой имеют тип `double`.

Если заданный по умолчанию тип в C# не тот, который нужен для литерала, можно явно определить его тип указанием суффикса. Чтобы определить литерал типа `long`, добавьте в конец *l* или *L*. Например, 12 имеет тип `int`, но `12L` имеет тип `long`. Чтобы определить значение целого числа без знака, добавьте в конец *u* или *U*. Таким образом, 100 имеет тип `int`, но 100U имеет тип `uint`. Чтобы определить целое число без знака типа `long`, используйте *ul* или *UL*. Например, 984375UL имеет тип `ulong`.

Чтобы указать литерал с плавающей точкой типа `float`, добавьте в конец *F* или *f*. Например, 10,19F имеет тип с плавающей точкой `float`. Хотя это и избыточно, вы можете определить литерал типа `double`, добавляя в конец *D* или *d*. (Как только что упоминалось, литералы с плавающей точкой по умолчанию имеют тип `double`.)

Чтобы определить литерал, представляющий десятичное число типа `decimal`, за его значением поставьте *m* или *M*. Например, 9,95M — десятичное число типа `decimal`, заданное в виде литерала.

Хотя целочисленные литералы обычно по умолчанию имеют тип `int`, `uint`, `long` или `ulong`, их все равно можно присваивать переменным типа `byte` (байт), `sbyte`, `short` или `ushort`, если присваиваемое значение может быть представлено типом той переменной, которой присваивается данное значение.

Шестнадцатеричные литералы

В программировании иногда легче использовать систему счисления с основанием 16, чем 10. Систему счисления с основанием 16 называют *шестнадцатеричной*, здесь используются цифры 0 до 9, плюс символы от A до F, которые обозначают 10, 11, 12, 13, 14 и 15. Например, шестнадцатеричное число 10 представляет 16 в десятичной системе. Из-за того что шестнадцатеричные числа используются часто, в C# можно определять целочисленный литерал в шестнадцатеричном формате. Шестнадцатеричный литерал должен начаться с 0x (нуль, за которым следует *x*). Вот некоторые примеры:

```
count = 0xFF; // 255 в десятичном представлении
incr = 0x1a; // 26 в десятичном представлении
```

Символьные управляющие последовательности

Включение символьных констант в одинарных кавычках работает для большинства печатных символов, но немногие символы, вроде перевода каретки, представляют специальную проблему, когда используется текстовый редактор. Кроме того, некоторые другие символы, такие как одинарные и двойные кавычки, имеют специальное значение в C#, так что их нельзя использовать непосредственно. По этим причинам в C# предусмотрены специальные *escape-последовательности*, показанные в табл. 2.2. Эти последовательности используются вместо символов, которые они представляют.

Например, следующая инструкция присваивает `ch` символ табуляции:

```
ch = '\t';
```

В следующем примере присваивается одинарная кавычка переменной ch:

```
ch = '\'';
```

Таблица 2.2. Символьные управляющие последовательности

Escape-последовательность	Описание
\a	Предупреждение (звонок)
\b	Backspace (Возврат на один символ)
\f	Form feed (Перевод страницы)
\n	Newline (Перевод строки)
\r	Перевод каретки
\t	Горизонтальная позиция табуляции
\v	Вертикальная позиция табуляции
\0	Null (Пустой указатель)
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Косая черта

Строковые литералы

C# поддерживает еще один тип литералов: строки. Строковый литерал — набор символов, заключенных в двойные кавычки. Например,

```
"это - испытание"
```

является строкой. Вы видели примеры строк во многих инструкциях `WriteLine( )` в предыдущих примерах программ.

В дополнение к обычным символам, строковый литерал может содержать также одну или несколько только что описанных escape-последовательностей. Рассмотрим, например, следующую программу. В ней используются escape-последовательности `\n` и `\t`.

Спросите у эксперта

ВОПРОС. Я знаю, что C++ позволяет определить целочисленные литералы в восьмеричной системе счисления. Позволяет ли C# задавать восьмеричные литералы?

ОТВЕТ. Нет. C# позволяет определять целочисленные литералы только в десятичной или шестнадцатеричной форме. Восьмеричная система счисления редко используется в современных средах программирования.

```
// Демонстрируются escape-последовательности в строках.

using System;

class StrDemo {
    static void Main() {
        // Используем \n, чтобы сгенерировать новую строку.
        Console.WriteLine("First line\nSecond line");
        // Используем табуляцию, чтобы выровнять вывод.
        Console.WriteLine("A\tB\tC");
        Console.WriteLine("D\tE\tF");
    }
}
```

Вывод приведен ниже:

```
First line
Second line
A       B       C
D       E       F
```

Обратите внимание, как используется escape-последовательность `\n`, чтобы перейти на новую строку. Вы не обязаны использовать несколько инструкций `WriteLine( )`, чтобы получить многострочный вывод. Просто вставьте `\n` в строку формата там, где нужно перейти на новую строку.

В дополнение к только что описанной форме строкового литерала, можно определить *дословный строковый литерал*. Дословный строковой литерал начинается со знака @, за которым следует строка в кавычках. Содержание строки в кавычках принимается без модификации и может охватить две или больше строк. Таким образом, можно включить знаки перехода на новую строку, позиции табуляции и т.д., но вы не должны использовать escape-последовательности. Единственное исключение — чтобы получить двойную кавычку ("), вы должны использовать две двойные кавычки в строке (""). Ниже приведена программа, которая демонстрирует дословные строковые литералы:

```
// Демонстрируем дословные строковые литералы.

using System;

class Verbatim {
    static void Main() {
        // Эта дословная строка в кавычках содержит
        // знак перехода на новую строку.
        Console.WriteLine(@"This is a verbatim
string literal
that spans several lines. ");
        // Эта строка содержит
        // также знаки табуляции.
        Console.WriteLine(@"Here is some tabbed output:
1    2    3    4
5    6    7    8
");
        Console.WriteLine(@"Programmers say, ""I like C#.""");
    }
}
```

Распечатка, полученная после прогона этой программы, приведена ниже:

```
This is a verbatim string literal
that spans several lines.

Here is some tabbed output:
1    2    3    4
5    6    7    8

Programmers say, "I like C#."
```

Важно обратить внимание на то, что в предыдущей программе дословные строковые литералы отображены точно так, как они введены в программу.

Преимущество дословного строкового литерала состоит в том, что можно определить вывод в программе точно так, как он появится на экране. Однако, в случае многострочных строк, произойдет перенос отступов из вашей программы, и вывод может быть несколько неожиданным. По этой причине программы в этой книге только ограниченно используют дословные строковые литералы. Но они все равно дают замечательную выгоду во многих ситуациях, связанных с форматированием.

Более подробное изучение переменных

Переменные были представлены в главе 1 “Основные принципы C#”. Вы узнали, что переменные объявляются в инструкции, которая имеет следующую форму:

тип имя-переменной;

где *тип* — это тип данных, хранимых в переменной, а *имя-переменной* — ее название (имя). Вы можете объявить переменную любого допустимого типа, включая только что описанные типы значения. Важно понять, что возможности переменной определены ее типом. Например, переменную типа `bool` нельзя использовать, чтобы сохранить значения с плавающей точкой. Кроме того, тип переменной не изменяется в течение всей ее жизни. Например, переменная типа `int` не может превратиться в переменную типа `char` (символ).

Все переменные в программах на C# должны быть объявлены. Это необходимо, потому что компилятор должен знать, какие данные переменная содержит, прежде чем он сможет должным образом скомпилировать любую инструкцию, в которой используется данная переменная. Это также дает возможность в C# выполнять строгий контроль соответствия типов.

В C# определено несколько различных видов переменных. Виды, которые мы использовали, называют *локальными переменными*, потому что они объявлены в пределах метода.

Спросите у эксперта

ВОПРОС. Предположим, что строка состоит из единственного (отдельного) символа. Тогда это то же самое, что и символ-литерал? Например, является ли “k” тем же самым, что и ‘k’?

ОТВЕТ. Нет. Вы не должны путать строки с символами. Литерал-символ представляет одну букву типа `char` (символ). Строка, содержащая только один символ, — все равно строка. Хотя строки состоят из символов, они имеют не тот же самый тип.

Инициализация переменных

Как вы уже заметили, один способ придать переменной значение заключается в том, чтобы написать оператор присваивания. Другой способ состоит в том, чтобы дать ей начальное значение в ее объявлении. Чтобы сделать это, за названием (именем) переменной поставьте знак = и присваиваемое значение. Общая форма инициализации приведена ниже:

тип имя-переменной = значение;

Здесь *значение* — это то значение, которое дается переменной *имя-переменной* при создании переменной *имя-переменной*. Значение должно быть совместимо с указанным типом.

Вот некоторые примеры:

```
int count = 10; // начальное значение 10
ch = 'X'; // инициализируем ch символом X
float f = 1.2F; // f инициализируется значением 1.2
```

При объявлении двух или большего числа переменных того же самого типа, используя разделенный запятыми список, можно дать начальные значения одной или большему числу переменных. Например:

```
int a, b = 8, c = 19, d; // b и c имеют инициализаторы
```

В этом случае инициализированы только `b` и `c`.

Динамическая инициализация

Хотя в предыдущих примерах использовались только константы в качестве инициализаторов, в C# переменные можно инициализировать динамически, используя любое выражение, допустимое в том месте, в котором объявлена переменная. Например, ниже приведена короткая программа, которая вычисляет объем цилиндра, если дан радиус его основания и его высота:

```
// Демонстрируем динамическую инициализацию.

using System;

class DynInit {
    static void Main() {
        double radius = 4, height = 5; // радиус = 4, высота = 5;

        // Динамически инициализируем объем volume.
        // объем = 3.1416 * радиус * радиус * высота;
        double volume = 3.1416 * radius * radius * height;
        // Теперь объем volume
        // динамически инициализирован во время выполнения.

        Console.WriteLine("Volume is " + volume);
    }
}
```

Здесь объявлены три локальных переменных: радиус `radius`, высота `height` и объем `volume`. Первые две, радиус `radius` и высота `height`, инициализированы константами. Однако объем `volume` инициализируется динамически по формуле объема цилиндра. Ключевой пункт здесь — то, что выражение инициализации может использовать любой элемент, допустимый в точке инициализации, включая запросы к методам, другим переменным и литералы.

Переменные неявного типа

Как уже объяснялось, в C# все переменные должны быть объявлены. Обычно объявление включает тип переменной, например `int` или `bool`, за которым следует имя переменной. Однако в C# 3.0 можно позволить компилятору ре-

шать, какой тип переменной следует приписать, основываясь на значении, которое приписывается ей при инициализации. Такую переменную называют переменной неявного типа.

Переменная неявного типа объявляется с помощью ключевого слова `var`, причем она должна быть инициализирована. Компилятор использует тип инициализатора, чтобы определить тип переменной. Вот пример:

```
var pi = 3.1416;
```

Поскольку `pi` инициализировано литералом с плавающей точкой (этот тип является `double` по умолчанию), тип `pi` будет `double`. Если бы `pi` было объявлено так:

```
var pi = 3.1416M;
```

тогда `pi` имело бы тип десятичного числа типа `decimal`.

Следующая программа демонстрирует переменные неявного типа:

```
// Демонстрируем переменные неявного типа.

using System;

class ImpTypedVar {
    static void Main() {

        // Это переменные неявного типа.
        var pi = 3.1416; // переменная pi имеет тип double
        var radius = 10; // переменная радиус radius имеет тип int

        // Сообщения msg и msg2 - строковые типы.
        // Переменные неявного типа.
        var msg = "Radius: ";
        var msg2 = "Area: ";

        // area явно объявлена как double.
        double area;

        Console.WriteLine(msg2 + radius);
        area = pi * radius * radius;
        Console.WriteLine(msg + area);

        Console.WriteLine();

        radius = radius + 2;

        Console.WriteLine(msg2 + radius);
        area = pi * radius * radius;

        Console.WriteLine(msg + area);

        // Следующая инструкция не будет скомпилирована, потому что
        // радиус radius имеет тип int и ему нельзя присвоить
        // значение с плавающей точкой.
```

```
        // radius = 12.2; // Ошибка!
    }
}
```

Вывод приведен ниже:

```
Area: 10
Radius: 314.16

Area: 12
Radius: 452.3904
```

Важно подчеркнуть, что переменная неявного типа — все равно переменная со строгим контролем типов. Обратите внимание, что следующая инструкция (строка) в программе закомментирована:

```
// radius = 12.2;      // Ошибка!
```

Это присваивание недопустимо, потому что радиус `radius` имеет тип `int`. Таким образом, ему нельзя присвоить значение с плавающей запятой. Единственное различие между переменной неявного типа и "обычной" переменной с явно заданным типом — то, как определяется тип. Как только тот тип определен, переменная имеет данный тип и этот тип установлен на всю жизнь переменной. Таким образом, тип радиуса `radius` нельзя изменить во время выполнения программы.

Переменные неявного типа добавили в C#, чтобы можно было справиться с несколькими специальными случаями, наиболее важные из которых касаются интегрированного языка запросов LINQ, который описан в этой книге позже. Что касается большинства переменных, то они должны иметь явные типы, потому что они делают код более легким для чтения и его проще понять. Переменные неявного типа должны использоваться только при необходимости. Они вообще не предназначены для замены нормальных объявлений переменных. В сущности, используйте, но не злоупотребляйте этой новой особенностью C#.

Последний момент: только одна переменная неявного типа может быть объявлена в любой момент. Поэтому следующее объявление

```
var count = 10, max = 20; // Ошибка!
```

является неправильным и не будет компилироваться, потому что оно пытается объявить `count` и `max` в то же самое время.

Область видимости и время существования переменных

Пока все переменные, которые мы использовали, были объявлены в начале метода `Main( )`. Однако C# позволяет объявить локальную переменную в пределах любого блока. Как уже упоминалось в главе 1 "Основные принципы C#", блок начинается с открывающейся фигурной скобки и заканчивается закрывающейся фигурной скобкой. Блок определяет *область видимости*. Таким

образом, каждый раз, когда вы начинаете новый блок, вы создаете новую область видимости. Область видимости определяет, какие имена видимы другим частям вашей программы. Он также определяет время существования локальных переменных.

Самые важные области видимости в C# — это те, которые определены классами, и те, которые определены в методах. Обсуждение области видимости класса (и переменных, объявленных в классах) откладывается до рассмотрения описания классов. Пока мы исследуем только области видимости, определенные в методах или в частях методов.

Область видимости, определенная методом, начинается с его открывающейся фигурной скобки и заканчивается его закрывающейся фигурной скобкой. Однако если метод имеет параметры, они также находятся в области видимости, определенной методом.

Как общее правило, локальные переменные, объявленные в некоторой области видимости, не видимы для кода, который определен вне этой области видимости. Таким образом, когда вы объявляете переменную в области видимости, вы предохраняете ее от доступа или изменения кодом, находящимся вне данной области видимости. И действительно, правила области видимости обеспечивают основу для инкапсуляции.

Области видимости могут быть вложены. Например, каждый раз, когда вы создаете блок кода, вы создаете новую, вложенную область видимости. Когда это происходит, внешняя область видимости включает внутреннюю область видимости. Это означает, что локальные переменные, объявленные во внешней области видимости, видимы для кода в пределах внутренней области видимости. Однако обратное неверно. Локальные переменные, объявленные в пределах внутренней области видимости, не будут видимы вне ее.

Чтобы понять действие вложенных областей видимости, рассмотрим следующую программу:

```
// Демонстрируем область действия блока.

using System;

class ScopeDemo {
    static void Main() {
        int x; // доступно всему коду в пределах Main( )
        x = 10;
        if (x == 10)
        { // начинаем новую область видимости
            int y = 20; // доступно только этому блоку
            // x и y оба доступны здесь.
            Console.WriteLine("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Ошибка! y здесь не доступен
        // Здесь y вне его области видимости.

        // x все еще доступен здесь.
```

```
        Console.WriteLine("x is " + x);
    }
}
```

Как указывают комментарии, переменная x объявлена в начале области видимости Main( ) и доступна для всего последующего кода в пределах Main( ). В пределах блока if объявлен y. Так как блок определяет область видимости, y видим только другому коду в пределах его блока. По этой причине находящаяся вне блока видимости y строка y = 100; закомментирована. Если удалить ведущий символ комментария, во время компиляции произойдет ошибка, потому что y не видим вне его блока. В пределах блока if может использоваться x, потому что код в пределах блока (т.е. вложенная область видимости) имеет доступ к переменным, объявленным в охватывающей области видимости.

В пределах блока локальную переменную можно объявить в любом месте, но обращение к ней допустимо только после того, как она объявлена. Таким образом, если вы определяете переменную в начале метода, она доступна всему коду в пределах данного метода. Наоборот, объявление переменной в конце блока фактически бесполезно, потому что никакой код не имеет доступа к ней.

Если в объявлении переменной имеется инициализатор, то переменная будет повторно инициализирована каждый раз при входе в блок, в котором содержится это объявление. Рассмотрим, например, следующую программу:

```
// Демонстрируем время существования переменной.

using System;

class VarInitDemo {
    static void Main() {
        int x;

        for (x = 0; x < 3; x++)
        {
            int y = -1; // y инициализируется каждый раз
                        // при входе в блок
            Console.WriteLine("y is: " + y); // это всегда печатает -1
            y = 100;
            Console.WriteLine("y is now: " + y);
        }
    }
}
```

Вывод, сгенерированный этой программой, приведен ниже:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

Как видно, y всегда повторно инициализируется значением –1 каждый раз при входе в блок. Даже при том, что впоследствии ему присваивается значение 100, это значение будет потеряно.

Есть одна причуда в правилах C# для областей видимости, которая удивит вас. Хотя блоки можно вложить, никакая переменная, объявленная во внутренней области видимости, не может иметь то же самое имя, что и переменная, объявленная в охватывающей области видимости. Например, следующая программа, в которой программист пробует объявить две отдельные переменные с тем же самым именем, не будет компилироваться:

```
/*
Эта программа пытается объявить переменную во внутренней области видимости с
таким же именем, что и переменная, объявленная во внешней области видимости.
*** Эта программа не будет компилироваться. ***
*/

using System;

class NestVar {
    static void Main() {
        int count;

        for (count = 0; count < 10; count = count + 1)
        {
            Console.WriteLine("This is count: " + count);

            // Правонарушитель!!! Это находится в
            // противоречии с предыдущим count.
            int count;
            // Нельзя объявить count снова, потому что
            // оно уже объявлено в Main( ).

            for (count = 0; count < 2; count++)
                Console.WriteLine("This program is in error!");
        }
    }
}
```

Если вы хорошо знаете C/C++, то вы хорошо знаете, что нет никакого ограничения на имена, которые вы даете переменным, объявленным во внутренней области видимости. Таким образом, в C/C++ объявление `count` в блоке внешнего цикла `for` полностью правильно (и допустимо). Однако в C/C++ такое объявление скрывает внешнюю переменную `count`. Проектировщики C# чувствовали, что такой тип сокрытия имени мог бы легко привести к ошибкам при программировании, и отвергли его.

Операторы

C# имеет богатый набор операторов. Оператор — символ, который указывает компилятору, какое математическое или логическое действие нужно выполнить с данными. C# имеет четыре общих класса операторов: *арифметические*, *поразрядные*, *отношений* и *логические*. C# также имеет несколько других операторов, которые обрабатывают некоторые специальные ситуации. В этой главе мы из-

учим арифметические и логические операторы, а также операторы отношений. Кроме того, мы изучим оператор присваивания. Поразрядные и другие специальные операторы будут рассмотрены позже.

Арифметические операторы

В C# определяются следующие арифметические операторы.

Оператор	Значение
+	Сложение
-	Вычитание (а также одноместный минус)
*	Умножение
/	Деление
%	Остаток от деления целых (модуль)
++	Приращение, увеличение, или инкремент
--	Уменьшение, или декремент

Операторы +, –, * и / работают так, как от них и ожидается. Они могут быть применены к любым числовым данных встроенного типа.

Хотя действия основных арифметических операторов известны всем читателям, % требует некоторого объяснения. Сначала вспомните, что когда / применяется к целым числам, любой остаток будет отброшен; например, 10/3 будет равняться 3 при целочисленном делении. Вы можете получить остаток от этого деления, используя оператор модуля %. Это делается в C# так же, как и в других языках. Это приводит к остатку от целочисленного деления. Например, 10% 3 равняется 1. В C# % может быть применен и к целому числу, и к типам с плавающей точкой. Таким образом, 10,0% 3,0 также равно 1. (Это отличается от языков C и C++, в которых операции модуля позволяются только на целочисленных типах.) Следующая программа демонстрирует оператор модуля:

```
// Демонстрировать оператор %.

using System;

class ModDemo {
    static void Main() {
        int iresult, irem; double dresult, drem;

        iresult = 10 / 3;
        irem = 10 % 3; // Используем оператор модуля.

        dresult = 10.0 / 3.0;
        drem = 10.0 % 3.0; // Используем оператор модуля.

        Console.WriteLine("Result and remainder of 10 / 3: " +
                          iresult + " " + irem);
        Console.WriteLine("Result and remainder of 10.0 / 3.0: " +
                          dresult + " " + drem);
```

```
    }
}
```

Распечатка, полученная после прогона программы, приведена ниже:

```
Result and remainder of 10 / 3: 3 1
Result and remainder of 10.0 / 3.0: 3.33333333333333 1
```

Как вы можете видеть, результатом операции % является остаток 1 и для целочисленных, и для чисел с плавающей точкой.

Приращение и декремент

Представленные в главе 1 “Основные принципы C#” операторы ++ и -- являются соответственно операторами приращения и декремента. Как вы увидите, они имеют несколько специальных свойств, которые делают их весьма интересными. Давайте начнем с рассмотрения того, что в точности делают операторы приращения и декремента.

Оператор приращения добавляет 1 к своему операнду, а оператор декремента вычитает 1. Поэтому

```
x = x + 1;
```

является тем же самым, что и

```
x++;
```

а

```
x = x - 1;
```

является тем же самым, что и

```
--x;
```

Операторы приращения и декремента могут или предшествовать (*префикс*) или следовать (*постфикс* или *суффикс*) операнду. Например:

```
x = x + 1;
```

можно написать как

```
++x; // префиксная форма
```

или как

```
x++; // постфиксная форма
```

В предшествующем примере нет никакого различия, применено ли приращение как префикс или постфикс. Однако, когда приращение или декремент используются как часть большего выражения, есть важное различие. В этом случае, когда оператор *предшествует* своему операнду, для получения результата операции используется значение операнда *после* приращения или декремента. Если оператор *следует за* его операндом, для получения результата операции используется значение операнда перед приращением или декрементом. Рассмотрим следующий фрагмент программы:

```
x = 10;
y = ++x;
```

В этом случае переменной y будет присвоено 11. Это потому, что x сначала увеличивается и затем используется его значение. Однако, если код написан как

```
x = 10;
y = x++;
```

то y будет присвоено 10. В этом случае значение сначала берется x, x увеличивается и затем возвращается первоначальное значение x. В обоих случаях x все равно присваивается 11; различие в том, что возвращается оператором.

Вот немного более сложный пример:

```
x = 10;
y = 2;
z = x++ - (x * y);
```

Это присваивает z значение –12. Причина состоит в том, что, когда вычисляется x++, x получает значение 11, но в дальнейшем вычислении все еще используется значение 10 (первоначальное значение x). Это означает, что z присваивается 10 – (11 * 2), что равно 12. Однако, если написать вот так:

```
z = ++x - (x * y);
```

то результат будет равен –11. Это потому что x сначала увеличивается и затем используется это его новое значение. Таким образом, z присваивается 11 – (11 * 2), что равно –11. Как показывает этот пример, возможность указать, когда следует выполнить операцию приращения или декремента, может дать существенные преимущества.

Операторы отношений и логические

Говоря об *операторах отношений* и *логических операторах*, следует отметить, что операторы *отношений* отображают отношения между значения, а *логические* операторы выполняются над значениями, которые бывают истинными или ложными. Результаты операторов отношений могут быть истинными или ложными, поэтому они часто встречаются вместе с логическими операторами. По этой причине они обсуждаются здесь вместе.

Операторы отношений приведены ниже.

Оператор	Значение
==	Равно
!=	Не равно
>	Больше чем
<	Меньше чем
>=	Больше чем или равно
<=	Меньше чем или равно

Логические операторы приведены ниже.

Оператор	Значение
&	AND (И)
|	OR (ИЛИ)
^	XOR (exclusive OR) (Неэквивалентно, или исключительный ИЛИ)
||	OR (ИЛИ)
&&	AND (И)
!	NOT (НЕ)

Результат относительных и логических операторов представляет собой значение типа `bool`.

Вообще говоря, объекты можно сравнивать на равенство или неравенство, используя == и !=. Однако операторы сравнения <>, <= и >= могут быть применены только к тем типам, которые поддерживают отношения порядка. Поэтому все операторы отношений применяются ко всем числовым типам. Однако значения типа `bool` можно сравнивать только на равенство или неравенство, так как истинные и ложные значения не упорядочены. Например, выражение `true > false` (истина > ложь) не имеет никакого значения в C#.

У логических операторов операнды должны иметь тип `bool` и результат логической операции имеет тип `bool`. Логические операторы `&`, `|`, `^` и `!` поддерживают основные логические операции AND (И), OR (ИЛИ), XOR (неэквивалентно) и NOT (НЕ), результат их вычисляется согласно следующей истинностной таблице.

p	q	p & q	p | q	p ^ q	!p
False (Ложь)	False (Ложь)	False (Ложь)	False (Ложь)	False (Ложь)	True (Истина)
True (Истина)	False (Ложь)	False (Ложь)	True (Истина)	True (Истина)	False (Ложь)
False (Ложь)	True (Истина)	False (Ложь)	True (Истина)	True (Истина)	True (Истина)
True (Истина)	True (Истина)	True (Истина)	True (Истина)	False (Ложь)	False (Ложь)

Как показывает таблица, результат операции исключительного OR истинен, только когда точно один операнд истинен (принимает значение `true`).

Ниже приведена программа, которая демонстрирует несколько операторов отношений и логических:

```
// Демонстрируем операторы отношений и логические.

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("this won't execute");
        if (i >= j) Console.WriteLine("this won't execute");
        if (i > j) Console.WriteLine("this won't execute");

        b1 = true;
        b2 = false;
        if (b1 & b2) Console.WriteLine("this won't execute");
        if (!(b1 & b2)) Console.WriteLine("!(b1 & b2) is true");
        if (b1 | b2) Console.WriteLine("b1 | b2 is true");
        if (b1 ^ b2) Console.WriteLine("b1 ^ b2 is true");
    }
}
```

Вывод программы приведен ниже:

```
i < j
i <= j
i != j
!(b1 & b2) is true
b1 | b2 is true
b1 ^ b2 is true
```

Логические короткие операторы

C# имеет специальные *короткие* версии логических операторов AND (И) и OR (ИЛИ), которые могут использоваться для повышения эффективности кода. Чтобы понять, почему, рассмотрим следующий пример. В операции AND, если первый операнд ложен, результат является ложным, независимо от того, какое значение имеет второй операнд. В операции OR, если первый операнд истинен, результат операции истинен, независимо от значения второго операнда. Таким образом, в этих двух случаях нет никакой потребности вычислять второй операнд. Не вычисляя второй операнд, экономим время и создаем более эффективный код.

Короткий оператор AND обозначается `&&`, а короткий оператор OR — `||`. Как уже упоминалось, их нормальные аналоги обозначаются `&` и `|`. Единственное различие между нормальными и короткими версиями состоит в том, что нормальные операнды будут всегда вычислять каждый операнд, а короткие версии схемы вычисляют второй операнд, только когда это необходимо.

Ниже приведена программа, которая демонстрирует короткий оператор AND. Программа определяет, является ли значение в `d` делителем `n`. Она делает это, выполняя операцию вычисления модуля. Если остаток от `n / d` нулевой, то `d` является делителем. Однако, поскольку операция вычисления модуля вовлекает деление, используется И (AND) в короткой форме, чтобы предотвратить ошибку деления на нуль.

```
// Демонстрируем короткие операторы.

using System;
```

```
class SCops {
    static void Main() {
        int n, d;

        n = 10;
        d = 2;

        // Здесь d = 2, так что операция модуля законна.
        if (d != 0 && (n % d) == 0)
            Console.WriteLine(d + " is a factor of " + n);

        d = 0; // теперь d обнулено

        // Поскольку d равно нулю, второй операнд не вычисляется.
        if (d != 0 && (n % d) == 0)
        // Короткий оператор предотвращает деление на нуль.
            Console.WriteLine(d + " is a factor of " + n);

        // Теперь, пробуем то же самое без короткого оператора.
        // Это вызовет ошибку деления на ноль.
        if (d != 0 & (n % d) == 0)
            // Теперь вычисляются оба выражения
            // и происходит деление на нуль.
            Console.WriteLine(d + " is a factor of " + n);
    }
}
```

Чтобы предотвратить ошибку деления на нуль, условный оператор `if` сначала выясняет, равен ли `d` нулю. Если это так, короткий AND (И) останавливает дальнейшее вычисление и не исполняет деление для нахождения модуля. Таким образом, в первом испытании `d` равно 2 и операция нахождения модуля выполняется. Затем `d` присваивается значение нуль. Поэтому при второй проверке условие будет ложным и операция нахождения модуля пропускается, благодаря чему удается избежать ошибки деления на нуль. Наконец, пробуем нормальный оператор AND. Он заставляет вычислить оба операнда, а это ведет к ошибке во время выполнения программы, когда происходит деление на нуль.

Еще один момент: короткий AND (И) также называется *условным AND* (*И*), а короткий OR (ИЛИ) также называется *условным OR* (*ИЛИ*).

Попробуйте вот это. Отобразите истинностную таблицу логических операторов

Сейчас вы создадите программу, которая отображает таблицу истинности для логических операторов C#. Эта программа использует несколько особенностей, описанных в этой главе, включая символьные управляющие последовательности и логические операторы C#. Она также иллюстрирует различия в старшинстве между арифметическим оператором + и логическими операторами.

Шаг за шагом

1. Создайте новый файл по имени `LogicalOpTable.cs`.

2. Чтобы гарантировать, что столбцы выстроены в линию, используйте escape-последовательность `\t`, чтобы вставить позиции табуляции в каждую строку вывода. Например, следующая инструкция `WriteLine( )` отображает заголовок таблицы:

```
Console.WriteLine("P\tQ\tAND\tOR\tXOR\tNOT");
```

3. Для каждой последующей строки в таблице используйте позиции табуляции, чтобы должным образом позиционировать результат каждой операции под ее надлежащим заголовком.
4. Вот полная распечатка программы `LogicalOpTable.cs`. Введите ее прямо сейчас.

```
// Печатать таблицу истинности для логических операторов.

using System;

class LogicalOpTable {
    static void Main() {

        bool p, q;

        Console.WriteLine("P\tQ\tAND\tOR\tXOR\tNOT");
        p = true; q = true;
        Console.Write(p + "\t" + q + "\t");
        Console.Write((p & q) + "\t" + (p | q) + "\t");
        Console.WriteLine((p ^ q) + "\t" + (!p));

        p = true; q = false;
        Console.Write(p + "\t" + q + "\t");
        Console.Write((p & q) + "\t" + (p | q) + "\t");
        Console.WriteLine((p ^ q) + "\t" + (!p));

        p = false; q = true;
        Console.Write(p + "\t" + q + "\t");
        Console.Write((p & q) + "\t" + (p | q) + "\t");
        Console.WriteLine((p ^ q) + "\t" + (!p));

        p = false; q = false;
        Console.Write(p + "\t" + q + "\t");
        Console.Write((p & q) + "\t" + (p | q) + "\t");
        Console.WriteLine((p ^ q) + "\t" + (!p));
    }
}
```

5. Скомпилируйте и выполните программу. Будет отображена следующая таблица:

```
P       Q       AND     OR      XOR     NOT
True    True    True    True    False   False
True    False   False   True    True    False
False   True    False   True    True    True
False   False   False   False   False   True
```

6. Обратите внимание на круглые скобки, окружающие логические операции в инструкциях `Write( )` и `WriteLine( )`. Они необходимы из-за старшинства операторов в C#. Оператор + по старшинству выше, чем логические операторы.
7. Самостоятельно попробуйте изменить программу так, чтобы она использовала и отображала единицы и нули, а не true и false.

Спросите у эксперта

ВОПРОС. Поскольку в некоторых случаях короткие операторы более эффективны, чем их нормальные аналоги, почему в C# все равно есть нормальные операторы AND и OR?

ОТВЕТ. В некоторых случаях необходимо вычислить оба операнда операций AND и OR из-за нужных побочных эффектов. Рассмотрим следующую программу:

```
// Побочные эффекты могут быть важны.

using System;

class SideEffects {
    static void Main() {
        int i;

        i = 0;

        // Здесь i увеличивается даже при том, что
        // условие условного оператора ложно.
        if (false & (++i < 100))
            Console.WriteLine("this won't be displayed");
        Console.WriteLine("if statement executed: " + i);
        // отображает 1

        // В этом случае i не увеличивается потому, что
короткий
        // оператор пропускает приращение.
        if (false && (++i < 100))
            Console.WriteLine("this won't be displayed");
        Console.WriteLine("if statement executed: " + i);
        // все еще 1 !!
    }
}
```

Как указывают комментарии, в первом условном операторе `i` увеличивается независимо от того, истинно или ложно условие. Однако, когда используется короткий оператор, переменная `i` не увеличивается, если первый операнд является ложным. Урок здесь в том, что если код ожидает выполнения правого операнда операции AND или OR, который нужно вычислить, то необходимо использовать нормальные формы этих операций.

Оператор присваивания

Вы использовали оператор присваивания, начиная с главы 1 “Основные принципы C#”. Теперь пришло время разобрать его форму. Оператор присваивания — единственный (отдельный) знак =. Оператор присваивания работает в C# очень похоже на то, как он делает это в других машинных языках. Он имеет следующую общую форму:

```
переменная-величина = выражение;
```

Здесь тип *переменной-величины* должен быть совместим с типом *выражения*.

Оператор присваивания действительно имеет один интересный атрибут, для вас незнакомый: он позволяет создавать цепочку присваиваний. Рассмотрим, например, следующий фрагмент:

```
int x, y, z;
x = y = z = 100; // присваивает x, y и z значение 100
```

Этот фрагмент устанавливает значения переменных `x`, `y` и `z`, равными 100, используя единственную инструкцию. Это работает, потому что оператор = имеет значение, равное присваиваемому значению. Таким образом, значение выражения `z = 100` равно 100, которое затем присваивается `y`, которое, в свою очередь, присваивается `x`. Использование “цепочек присваивания” является простым способом присвоить группе переменных общее значение.

Составные присваивания

C# обеспечивает специальные составные операторы присваивания, которые упрощают кодирование некоторых операторов присваивания. Давайте начнем с примера. Оператор присваивания, показанный ниже:

```
x = x + 10;
```

можно записать, используя составное присваивание, так:

```
x += 10;
```

Пара операторов += говорит, что компилятор должен присвоить `x` значение `x` плюс 10.

Вот другой пример. Инструкция

```
x = x - 100;
```

делает то же самое, что и

```
x -= 100;
```

Обе инструкции присваивают `x` значение `x` минус 100.

Есть составные операторы присваивания для многих бинарных операторов (т.е. тех, которые требуют двух операндов). Общая форма стенографии

переменная op= выражение;

Ниже перечислены арифметические и логические операторы присваивания.

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

Поскольку составные операторы присваивания короче, чем их несоставные эквиваленты, их также иногда называют стенографическими операторами присваивания.

Составные операторы присваивания дают две выгоды. Во-первых, они более компактны, чем их "обычные" эквиваленты. Во-вторых, они могут привести к более эффективному выполнимому коду (потому что левый операнд вычисляется только однажды). По этим причинам в профессионально составленных программах на C# часто используются составные операторы присваивания.

Преобразование типов в присваиваниях

В программировании часто приходится присвоить значение одного типа переменной другого типа. Например, иногда нужно присвоить значение `int` переменной с плавающей точкой, как показано ниже:

```
int i;
float f;

i = 10;
f = i; // присваивается значение типа int переменной с плавающей точкой (типа
float)
```

Когда совместимые типы смешаны в присваивании, значение правой части автоматически преобразовывается в тип левой части. Таким образом, в предыдущем фрагменте значение `i` преобразуется в тип с плавающей точкой (`float`) и затем присваивается `f`. Однако из-за строгого контроля соответствия типов в C#, не все типы совместимы и, таким образом, не все преобразования типов позволяются неявно. Например, `bool` и `int` несовместимы.

Когда один тип данных присваивается переменной другого типа, неявное преобразование типов будет иметь место автоматически, если:

- два типа совместимы.
- тип адресата более широкий, чем исходный тип.

Когда эти два условия выполнены, происходит преобразование расширения. Например, тип `int` всегда является достаточно большим, чтобы содержать все допустимые значения байта (`byte`), причем `int` и байт (`byte`) — целочисленные типы, так что можно применить неявное преобразование.

Для преобразований расширения числовые типы, включая целые числа и типы с плавающей точкой, являются совместимыми. Например, следующая программа совершенно правильна, поскольку расширяющее преобразование `long` в `double` выполняется автоматически:

```
// Демонстрируется неявное преобразование long в double.

using System;

class LtoD {
    static void Main() {
        long L;
        double D;

        L = 100123285L;
        D = L; // Автоматическое преобразование long в double.

        Console.WriteLine("L and D: " + L + " " + D);
    }
}
```

Хотя есть неявное преобразование `long` в `double`, нет никакого неявного преобразования от `double` в `long`, так как такое преобразование не расширяющее. Таким образом, следующая версия предыдущей программы недопустима:

```
// *** Эта программа не будет компилироваться. ***

using System;

class LtoD {
    static void Main() {
        long L;
        double D;

        D = 100123285.0;
        L = D; // Правонарушитель!!!
        // Нет автоматического преобразования из double в long.

        Console.WriteLine("L and D: " + L + " " + D);
    }
}
```

В дополнение к только что описанным ограничениям, нет никаких неявных преобразований между десятичными числами (тип `decimal`) и числами с плавающей точкой (тип `float`) или числами типа `double`, или числовых типов в символы (тип `char`), или булевы значения (тип `bool`). Кроме того, символы (тип `char`) и булевы значения (тип `bool`) не совместимы.

Приведение несовместимых типов

Хотя неявные преобразования типов полезны, они не удовлетворяют все потребности программирования, потому что они применяются только к расширяющим преобразованиям между совместимыми типами. Во всех других случаях необходимо использовать приведение. Приведение — команда компилятору, позволяющая конвертировать (преобразовать) выражение в указанный тип.

Таким образом, это запрос явного преобразования типов. Приведение имеет следующую общую форму:

(*целевой-тип*) *выражение*

Здесь *целевой-тип* определяет желательный тип, к которому преобразуется указанное выражение. Например, если тип выражения `x/y` необходимо преобразовать в `int`, нужно написать

```
double x, y;
// ...
(int) (x / y)
```

Здесь даже при том, что `x` и `y` имеют тип `double`, приведение конвертирует результат выражения к типу `int`. Круглые скобки, окружающие `x / y`, необходимы. В противном случае приведение к `int` применилось бы только к `x`, а не к результату деления. Приведение здесь необходимо, потому что нет никакого неявного преобразования из `double` в `int`.

Когда для приведения необходимо сужающее преобразование, информация иногда теряется. Например, при приведении `long` в `int` информация будет потеряна, если значение типа `long` больше, чем диапазон `int`, потому что старшие биты будут удалены. Когда значение с плавающей точкой приводится к целочисленному типу, дробная часть будет также потеряна из-за усечения. Например, если значение 1,23 присваивается целочисленной переменной, в конце концов будет присвоено просто значение 1. Дробная часть 0,23 будет потеряна.

Следующая программа демонстрирует некоторые преобразования типов, которые требуют приведений:

```
// Демонстрируем приведение.

using System;

class CastDemo {
    static void Main() {
        double x, y;
        byte b;
        int i;
        char ch;

        x = 10.0;
        y = 3.0;

        i = (int)(x / y); // В этом преобразовании произойдет усечение.
        Console.WriteLine("Integer outcome of x / y: " + i);

        i = 100;
        b = (byte)i; // Никакой потери информации здесь нет.
                     // Байт может содержать значение 100.
        Console.WriteLine("Value of b: " + b);

        i = 257;
        b = (byte)i; // На сей раз информация потеряна.
```

```
        // Байт (тип byte) не может содержать значение 257.
        Console.WriteLine("Value of b: " + b);

        b = 88; // ASCII-код X
        ch = (char)b;
        // Приведение между несовместимыми типами.
        Console.WriteLine("ch: " + ch);
    }
}
```

Распечатка, полученная после прогона программы, приведена ниже:

```
Integer outcome of x / y: 3
Value of b: 100
Value of b: 1
ch: X
```

В программе приведение (`x/y`) к `int` приводит к усечению дробной части и информация теряется. Затем, когда `b` присваивается значение 100, потери информации не происходит, потому что байт может содержать значение 100. Однако, когда делается попытка присвоить `b` значение 257, происходит потеря информации, потому что 257 превышает диапазон байта. Это приводит к наличию в `b` значения 1, потому что только 1 бит установлен в младших 8 битах двоичного представления 257. Наконец, никакая информация не теряется, но приведение необходимо при присваивании значения байта символу.

Старшинство операторов

В табл. 2.3 показан порядок выполнения (старшинство) всех операторов C#, начиная от наиболее высокого до наиболее низкого. Эта таблица включает несколько операторов, которые будут обсуждены позже в этой книге.

Преобразование типов в выражениях

В выражениях можно смешивать два или больше данных различных типов, если они совместимы. Например, в выражениях можно смешивать данные типов `short` и `long`, потому что они оба представляют собой числовые типы. Когда различные типы данных смешаны в выражении, они по очереди преобразовываются в тот же самый тип, когда над ними выполняются операции.

Преобразования выполняются по правилам продвижения типа в C#. Вот алгоритм, который определяет правила для двоичных операций:

```
ЕСЛИ один операнд является десятичным (decimal), ТО другой операнд
продвигается к десятичному числу (decimal)
(только если он не имеет типа с плавающей точкой (float) или double, в этом
случае фиксируется ошибка).

ИНАЧЕ ЕСЛИ один из операндов имеет тип double, второй продвигается к double.
ИНАЧЕ ЕСЛИ один операнд представляет собой операнд с плавающей точкой (float),
```

второй продвигается к типу с плавающей точкой (**float**).
ИНАЧЕ ЕСЛИ один операнд имеет тип **ulong**, второй продвигается к **ulong** (если он не имеет тип **sbyte**, **short**, **int** или **long**, в этом случае фиксируется ошибка).
ИНАЧЕ ЕСЛИ один операнд имеет тип **long**, второй продвигается к **long**.
ИНАЧЕ ЕСЛИ один операнд имеет тип **uint** и второй имеет тип **sbyte**, **short** или **int**, оба продвигаются к **long**.
ИНАЧЕ ЕСЛИ один операнд имеет тип **uint**, второй продвигается к **uint**.
ИНАЧЕ оба операнда продвигаются к **int**.

Таблица 2.3. Старшинство операторов в C#

Наиболее высокий									
() [] unchecked (без контроля типов)	.	++(постфикс)	--(постфикс)	checked	new	sizeof	typeof		
!	~	(cast)	+(одноместный)	-(одноместный)		++(префикс)	--(префикс)		
*	/	%							
+	-								
<<	>>								
<	>	<=	>=	is					
==	!=								
&									
^									
|									
&&									
||									
??									
?:									
=	op=	=>							
Наиболее низкий									

Есть несколько важных моментов, которые следует отметить в правилах продвижения типа. Во-первых, не все типы можно смешивать в выражении. Определенно, нет никакого неявного преобразования от типа с плавающей точкой (`float`) или от `double` к десятичному числу (`decimal`) и нельзя смешать `ulong` с любым типом целого числа со знаком. Чтобы смешивать эти типы, требуется использовать явное приведение.

Во-вторых, обратите особое внимание на последнее правило. Оно заявляет, что если ни одно из предыдущих правил не применяется, то все другие операнды продвигаются к `int`. Поэтому при вычислении выражений значения всех данных типов `char`, `sbyte`, `byte`, `ushort` и `short` продвигаются к `int`. Это называют *целочисленным продвижением*. Это также означает, что результат всех арифметических операций будет не меньше, чем `int`.

Важно понять, что продвижения типа относятся только к значениям, над которыми выполняются операции во время вычисления выражения. Например, если значение переменной типа байт (`byte`) продвигается к `int` в выражении, то вне этого выражения переменная все равно имеет тип байт (`byte`). Продвижение типов затрагивает только вычисление выражений.

Продвижение типа может, однако, привести к несколько неожиданным результатам. Например, когда арифметическая операция выполняется над двумя байтовыми (типа `byte`) значениями, происходит следующая последовательность действий: сначала байтовые (типа `byte`) операнды продвигаются к `int`. Затем выполняется операция, которая приводит к результату типа `int`. Таким образом, результат операции двумя байтовыми (типа `byte`) значениями будет иметь тип `int`. Это совсем не то, что можно было бы ожидать интуитивно. Рассмотрим следующую программу:

```
// Удивительное и неожиданное при продвижениях типа!

using System;

class PromDemo {
    static void Main() {
        byte b;
        int i;

        b = 10;
        // Никакого приведения не нужно, потому что
        // результат вычисления уже приведен к int.
        i = b * b; // OK, приведение не нужно

        b = 10;
        // А здесь приведение необходимо, чтобы присвоить
        // значение типа int байтовой переменной (тип byte).
        b = (byte)(b * b); // приведение необходимо!!

        Console.WriteLine("i and b: " + i + " " + b);
    }
}
```

Несколько противоинтуитивно, что никакого приведения не нужно при присваивании `b * b` переменной `i`, потому что `b` уже будет продвинуто к `int` при вычислении выражения. Таким образом, тип результата `b * b` будет `int`. Однако, чтобы присвоить `b * b` переменной `b`, действительно нужно приведение обратно к байту (типу `byte`)! Помните об этом, когда получаете неожиданные сообщения об ошибках несовместимости типов в выражениях, которые выглядят, казалось бы, совершенно идеально.

Такая же ситуация происходит при выполнении операций над символами. Например, в следующем фрагменте приведение обратно к символу (к типу `char`) необходимо из-за продвижения `ch1` и `ch2` к `int` в выражении:

```
char ch1 = 'a', ch2 = 'b';

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

Без приведения тип результата добавления `ch1` к `ch2` был бы `int`, а значение этого типа нельзя присваивать символу (переменной типа `char`).

Приведения полезны не только при преобразовании типов в присваивании. Они могут использоваться и в выражениях. Рассмотрим, например, следующую

программу. В ней используется приведение к double, чтобы получить дробную часть от деления, которое в противном случае интерпретировалось бы как целочисленное.

```
// Использование приведения.

using System;

class UseCast {
    static void Main()  {
        int i;

        for (i = 1; i < 5; i++)
        {
            Console.WriteLine(i + " / 3: " + i / 3);
            Console.WriteLine(i + " / 3 with fractions: {0:#.##}",
            (double)i / 3); Console.WriteLine();
        }
    }
}
```

В выражении:

```
(double) i / 3
```

приведение к double преобразует i к типу double, что гарантирует, что дробная часть от деления на 3 сохраняется. Распечатка, полученная после прогона программы, приведена ниже:

```
1 / 3: 0
1 / 3 with fractions: .33

2 / 3: 0
2 / 3 with fractions: .67

3 / 3: 1
3 / 3 with fractions: 1

4 / 3: 1
4 / 3 with fractions: 1.33
```

Спросите у эксперта

ВОПРОС. Происходят ли продвижения типа в одноместных операциях, например в одноместном –?

ОТВЕТ. Да. При выполнении одноместных операций операнды, меньшие, чем **int** (т.е. типы байт (**byte**), **sbyte**, **short** и **ushort**) продвигаются к **int**. Кроме того, операнд типа символ (**char**) преобразовывается в **int**. Кроме того, если значение **uint** инвертируется, оно продвигается к **long**.

Интервалы и круглые скобки

Выражение в C# может содержать позиции табуляции и пробелы, чтобы сделать его более удобочитаемым. Например, следующие два выражения задают те же самые действия, но второе более легкое для чтения:

```
x=10/y*(127-x);
x = 10 / y * (127 - x);
```

Круглые скобки могут использоваться для группировки подвыражений, таким образом они фактически увеличивают старшинство заключенных в них операций точно так же, как и в алгебре. Использование избыточных или дополнительных круглых скобок не вызывает ошибок или замедления выполнения выражения. Я рекомендую использовать круглые скобки, чтобы прояснить, как выражение вычисляется, и для вас непосредственно, и для других программистов, которым, вероятно, придется поддерживать вашу программу позже. Например, какое из следующих двух выражений более легкое для чтения:

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

Попробуйте вот это. Вычислите регулярные платежи по ссуде

Как уже упоминалось, десятичный тип (`decimal`) особенно хорошо подходит для вычислений с деньгами. Следующая программа демонстрирует использование этой возможности. Программа вычисляет регулярные платежи по ссуде, например по автомобильной ссуде. Зная стартовый баланс, отрезок времени, количество платежей в год и процентную ставку, программа вычисляет оплату. Поскольку это является финансовым вычислением, для представления данных имеет смысл использовать именно десятичный тип `decimal`. Эта программа также демонстрирует приведения и другие методы из библиотеки C#.

Чтобы вычислить платежи, используется следующая формула оплаты:

$$\text{Payment} = \frac{\text{IntRate} * \frac{\text{Principal}}{\text{PayPerYear}}}{1 - \left(\frac{\text{IntRate}}{\text{PayPerYear}} + 1\right)^{-\text{PayPerYear} * \text{NumYears}}},$$

где IntRate определяет процентную ставку, Principal содержит стартовый баланс, PayPerYear — количество платежей в год и NumYears определяет длительность ссуды в годах.

Обратите внимание, что в знаменателе формулы необходимо возвести одно значение в степень, задаваемую другим значением. Чтобы сделать это, необходимо использовать имеющийся в C# математический метод `Math.Pow( )`. Вот как он вызывается:

result = `Math.Pow(`*base*`,` *exp*`);`

Метод `Pow( )` возвращает значение *base*, возведенное в степень *exp*. Параметры `Pow( )` должны иметь тип `double`, и этот метод возвращает значение типа `double`. Это означает, что придется использовать приведение, чтобы выполнить преобразования между `double` и десятичным `decimal`.

Шаг за шагом

1. Создайте новый файл по имени RegPay.cs.
2. Ниже перечислены переменные, которые будут использоваться программой:

```
decimal Principal;    // десятичный первоначальный баланс
decimal IntRate;      // процентная ставка как
                      // десятичное число decimal, например 0.075
decimal PayPerYear;   // количество платежей в год - десятичное число
decimal NumYears;     // количество лет - десятичное число
decimal Payment;      // регулярная плата
decimal numer, denom; // рабочие переменные - десятичные числа
double b, e;          // основание и показатель степени
                      // для вызова Pow()
```

3. Большинство вычислений выполняется с данными десятичного типа (`decimal`), поэтому и большинство переменных имеет тип десятичных чисел (`decimal`).
4. Обратите внимание, что объявление каждой переменной сопровождается комментарием, который описывает ее использование. Это при чтении программы помогает понять назначение каждой переменной. Хотя в этой книге для коротких программ мы не будем включать такие детальные комментарии, это хорошая практика, и ей нужно следовать, поскольку программы становятся все более длинными и все более сложными.
5. Добавьте следующие строки программы, которые определяют информацию по ссуде. В нашем случае, первоначальный баланс `Principal` равен 10 000 долл., процентная ставка `IntRate` = 7,5%, количество платежей в год `PayPerYear` равно 12 и длительность ссуды `NumYears` 5 лет.

```
Principal = 10000.00m;
IntRate = 0.075m;
PayPerYear = 12.0m;
NumYears = 5.0m;
```

6. Добавьте строки, которые выполняют финансовое вычисление:

```
numer = IntRate * Principal / PayPerYear;

e = (double) -(PayPerYear * NumYears);
b = (double) (IntRate / PayPerYear) + 1;

denom = 1 - (decimal) Math.Pow(b, e);

Payment = numer / denom;
```

7. Обратите внимание, как должны использоваться приведения, чтобы передать значения `Pow( )` и конвертировать возвращаемое значение. Помните, в C# нет никаких неявных преобразований между десятичным числом (тип `decimal`) и `double`.
8. Закончите программу выводом регулярных платежей, как показано ниже:

```
Console.WriteLine("Payment is {0:C}", Payment);
```

9. Вот полная распечатка программы `RegPay.cs`:

```
// Вычислить регулярные платежи за ссуду.

using System;

class RegPay {
    static void Main() {
        decimal Principal;    // десятичный первоначальный баланс
        decimal IntRate;      // процентная ставка как десятичное
                              // число decimal, например 0.075
        decimal PayPerYear;   // количество платежей в год -
                              // десятичное число
        decimal NumYears;     // количество лет - десятичное число
        decimal Payment;       // регулярная плата
        decimal numer, denom; // рабочие переменные - десятичные числа
        double b, e;          // основание и показатель степени
                              // для вызова Pow()

        Principal = 10000.00m;
        IntRate = 0.075m;
        PayPerYear = 12.0m;
        NumYears = 5.0m;

        numer = IntRate * Principal / PayPerYear;

        e = (double)-(PayPerYear * NumYears);
        b = (double)(IntRate / PayPerYear) + 1;

        denom = 1 - (decimal)Math.Pow(b, e);

        Payment = numer / denom;

        Console.WriteLine("Payment is {0:C}", Payment);
    }
}
```

Вот вывод из программы:

```
Payment is $200.38
```

Перед тем как двигаться дальше, попробуйте заставить эту программу вычислить регулярные платежи при других значениях ссуды, периодичности и процентных ставок.

Глава 2. Самопроверка

1. Почему в C# строго определен диапазон и поведение его простых типов?
2. Чем является в C# символьный тип и как он отличается от символьного типа в некоторых других языках программирования?
3. Значение `bool` может иметь любое значение, которое вы любите, потому что любое значение, отличное от нуля, представляет собой истину `true`. Истинно или ложно это утверждение?
4. Пусть дан текст:

```
One
Two
Three
```

5. Используйте единственную строку и escape-последовательности, чтобы записать инструкцию `WriteLine( )`, которая вывела этот текст.
6. Что не так в этом фрагменте:

```
for(i = 0; i < 10; i++) {
int sum;

sum = sum + i;
}
Console.WriteLine("Sum is: " + sum);
```

7. Объясните различие между префиксными и постфиксными формами оператора приращения.
8. Покажите, как короткий AND (И) может использоваться для предотвращения ошибки деления на нуль.
9. К каким типам продвигаются типы байт (`byte`) и `short` в выражении?
10. Какой из следующих типов нельзя смешивать в выражении с десятичным значением (типом `decimal`)?
    A. `float` (с плавающей точкой)
    B. `int`
    C. `uint`
    D. `byte` (байт)
11. Вообще говоря, когда необходимо приведение?
12. Напишите программу, которая находит все простые числа между 2 и 100.
13. Самостоятельно перепишите программу, которая генерирует таблицу истинности. (Эта программа приведена в разделе “Попробуйте вот это. Отобразите истинностную таблицу логических операторов”.) Отобразите таблицу истинности логических операторов так, чтобы в программе использовались дословные строковые литералы с позицией табуляции и символами перехода на новую строку (newline), а не escape-последовательности.