## Знакомство с платформой .NET и средой Visual Studio .NET

Совокупность средств, с помощью которых программы пишут, корректируют, преобразуют в машинные коды, отлаживают и запускают, называют *средой разработки*, или *оболочкой*. Среда разработки обычно содержит:

* текстовый *редактор*, предназначенный для ввода и корректировки текста программы;
* *компилятор*, с помощью которого программа переводится с языка, на котором она написана, в машинные коды;
* средства *отладки и запуска программ* ;
* общие *библиотеки*, содержащие многократно используемые элементы программ;
* *справочную систему* и другие элементы.

Под *платформой* понимается нечто большее, чем среда разработки для одного языка. Платформа .NET (произносится "дотнет") включает не только среду разработки для нескольких языков программирования, называемую Visual Studio.NET, но и множество других средств, например, механизмы поддержки баз данных, электронной почты и коммерции.

Среда разработки Visual Studio.NET предоставляет мощные и удобные средства написания, корректировки, компиляции, отладки и запуска приложений, использующих .NET-совместимые языки. Корпорация Microsoft включила в платформу средства разработки для четырех языков: C#, VB.NET, C++ и J#.

Платформа .NET является *открытой средой*. Это значит, что компиляторы для нее могут поставляться и сторонними разработчиками. Для обеспечения переносимости компиляторы, входящие в состав платформы, переводят программу не в машинные коды, а в промежуточный язык (Common Intermediate Language, CIL, или просто IL), который не содержит команд, зависящих от языка, операционной системы и типа компьютера. Программа на этом языке выполняется не самостоятельно, а под управлением системы, которая называется *общеязыковой средой выполнения* (Common Language Runtime, CLR).

Среда CLR может быть реализована для любой операционной системы. При выполнении программы CLR вызывает так называемый JIT-компилятор, переводящий код с языка IL в машинные команды конкретного процессора, которые немедленно выполняются. JIT означает "just in time", что можно перевести как "вовремя", то есть компилируются только те части программы, которые требуется выполнить в данный момент. Каждая часть программы компилируется один раз и сохраняется в кэше для дальнейшего использования.

Компилятор в качестве результата своего выполнения создает так называемую *сборку* — файл с расширением **exe** или **dll**, который содержит код на языке IL и метаданные. *Метаданные* представляют собой сведения об объектах, используемых в программе, а также сведения о самой сборке. Они позволяют организовать межъязыковое взаимодействие, обеспечивают безопасность и облегчают *развертывание приложений*, то есть установку программ на компьютеры пользователей.

Во время работы программы среда CLR следит за тем, чтобы выполнялись только разрешенные операции, осуществляет распределение и очистку памяти и обрабатывает возникающие ошибки. Это многократно повышает безопасность и надежность программ.

Платформа .NET содержит огромную *библиотеку классов*, которые можно использовать при программировании на любом языке .NET. Подробное изучение библиотеки классов .NET — необходимая, но и наиболее трудоемкая задача программиста при освоении этой платформы. Библиотека классов вместе с CLR образуют *каркас* (framework), то есть основу платформы.

Все .NET-совместимые языки должны отвечать требованиям *общеязыковой спецификации* (Common Language Specification, CLS), в которой описывается набор общих для всех языков характеристик. Это позволяет использовать для разработки приложения несколько языков программирования и вести полноценную межъязыковую отладку. Все программы независимо от языка используют одни и те же базовые классы библиотеки .NET.

Приложение в процессе разработки называется *проектом*. Проект объединяет все необходимое для создания приложения: файлы, папки, ссылки и прочие ресурсы. Среда Visual Studio.NET позволяет создавать проекты различных типов, например:

* *Windows-приложение* использует элементы интерфейса Windows, включая формы, кнопки, флажки и прочее;
* *консольное приложение* выполняет вывод "на консоль", то есть в окно командного процессора;
* *библиотека классов* объединяет классы, которые предназначены для использования в других приложениях;
* *веб-приложение* — это приложение, доступ к которому выполняется через браузер (например, Internet Explorer) и которое по запросу формирует веб-страницу и отправляет ее клиенту по сети;

Несколько проектов можно объединить в *решение* (solution). Это облегчает совместную разработку проектов

Платформа .NET рассчитана на объектно-ориентированную технологию создания программ, поэтому прежде чем начинать изучение языка C#, необходимо познакомиться с основными понятиями объектно-ориентированного программирования (ООП).

## Введение в объектно-ориентированное программирование

Принципы ООП проще всего понять на примере программ моделирования. В реальном мире каждый предмет или процесс обладает набором статических и динамических характеристик, иными словами, свойствами и поведением. *Поведение объекта* зависит от его *состояния* и *внешних воздействий*. Например, объект "автомобиль" никуда не поедет, если в баке нет бензина, а если повернуть руль, изменится положение колес.

Понятие объекта в программе совпадает с обыденным смыслом этого слова: *объект представляется как совокупность данных*, *характеризующих его состояние*, *и функций их обработки*, *моделирующих его поведение*. Вызов функции на выполнение часто называют *посылкой сообщения* объекту. Например, вызов функции "повернуть руль" интерпретируется как посылка сообщения "автомобиль, поверни руль!".

При создании объектно-ориентированной программы предметная область представляется в виде совокупности объектов. Выполнение программы состоит в том, что объекты обмениваются сообщениями. Это позволяет использовать при программировании понятия, более адекватно отражающие предметную область.

При представлении реального объекта с помощью программного необходимо выделить в первом его существенные особенности. Их список зависит от цели моделирования. Например, объект "крыса" с точки зрения биолога, изучающего миграции, ветеринара или, скажем, повара будет иметь совершенно разные характеристики. Выделение существенных с той или иной точки зрения свойств называется *абстрагированием*. Таким образом, программный объект — это абстракция.

Важным свойством объекта является его обособленность. Детали реализации объекта, то есть внутренние структуры данных и алгоритмы их обработки, скрыты от пользователя объекта и недоступны для непреднамеренных изменений. Объект используется через его *интерфейс* — совокупность правил доступа.

Скрытие деталей реализации называется *инкапсуляцией* (от слова "капсула"). Ничего сложного в этом понятии нет: ведь и в обычной жизни мы пользуемся объектами через их интерфейсы. Сколько информации пришлось бы держать в голове, если бы для просмотра новостей надо было знать устройство телевизора!

Таким образом, объект является "черным ящиком", замкнутым по отношению к внешнему миру. Это позволяет представить программу в укрупненном виде — на уровне объектов и их взаимосвязей, а следовательно, управлять большим объемом информации и успешно отлаживать сложные программы.

Сказанное можно сформулировать более кратко и строго: *объект* — *это инкапсулированная абстракция с четко определенным интерфейсом*.

Инкапсуляция позволяет изменить реализацию объекта без модификации основной части программы, если его интерфейс остался прежним. Простота модификации является очень важным критерием качества программы: ведь любой программный продукт в течение своего жизненного цикла претерпевает множество изменений и дополнений.

Кроме того, инкапсуляция позволяет использовать объект в другом окружении и быть уверенным, что он не испортит не принадлежащие ему области памяти, а также создавать библиотеки объектов для применения во многих программах. Инкапсуляция наряду с наследованием и полиморфизмом считаются тремя "китами", на которых стоит ООП.

Каждый год в мире пишется огромное количество новых программ, и важнейшее значение приобретает возможность многократного использования кода. Преимущество объектно-ориентированного программирования состоит в том, что для объекта можно определить наследников, корректирующих или дополняющих его поведение. При этом нет необходимости не только повторять исходный код родительского объекта, но даже иметь к нему доступ.

*Наследование* является мощнейшим инструментом ООП и применяется для следующих взаимосвязанных целей:

* исключения из программы повторяющихся фрагментов кода;
* упрощения модификации программы;
* упрощения создания новых программ на основе существующих.

Кроме того, только благодаря наследованию появляется возможность использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

Наследование позволяет создавать *иерархии объектов*. Иерархия представляется в виде дерева, в котором более общие объекты располагаются ближе к корню, а более специализированные — на ветвях и листьях. Наследование облегчает использование библиотек объектов, поскольку программист может взять за основу объекты, разработанные кем-то другим, и создать наследников с требуемыми свойствами.

Объект, на основании которого строится новый объект, называется *родительским* объектом, объектом-предком, базовым классом, или суперклассом, а унаследованный от него объект — *потомком*, подклассом, или производным классом.

ООП позволяет писать гибкие, расширяемые и читабельные программы. Во многом это обеспечивается благодаря *полиморфизму*, под которым понимается возможность во время выполнения программы с помощью одного и того же имени выполнять разные действия или обращаться к объектам разного типа. Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов.

Подводя итог сказанному, сформулирую *достоинства ООП*:

* использование при программировании понятий, близких к предметной области;
* возможность успешно управлять большими объемами исходного кода благодаря инкапсуляции, то есть скрытию деталей реализации объектов и упрощению структуры программы;
* возможность многократного использования кода за счет наследования;
* сравнительно простая возможность модификации программ;
* возможность создания и использования библиотек объектов.

Эти преимущества особенно явно проявляются при разработке программ большого объема и классов программ. Однако ничто не дается даром: создание объектно-ориентированной программы представляет собой весьма непростую задачу. Чтобы эффективно использовать готовые объекты из библиотек, необходимо освоить большой объем достаточно сложной информации. Неграмотное же применение ООП способно привести к созданию излишне сложных программ, которые невозможно отлаживать и усовершенствовать.

#### Понятие класса

Для представления объектов в языках C#, Java, С++, Delphi и т. п. используется понятие *класс*, аналогичное обыденному смыслу этого слова в контексте "класс членистоногих", "класс млекопитающих", "класс задач" и т. п. Класс является обобщенным понятием, определяющим характеристики и поведение некоторого множества конкретных объектов этого класса, называемых *экземплярами класса*.

"Классический" класс содержит *данные*, задающие *свойства объектов класса*, и *функции*, определяющие их поведение. В последнее время в класс часто добавляется третья составляющая — *события*, на которые может *реагировать объект* класса[**2)**](http://www.intuit.ru/department/pl/phlcsharp/1/footnote.1.2.htm).

Все классы библиотеки .NET, а также все классы, которые создает программист в среде .NET, имеют одного общего предка — класс object, и организованы в единую иерархическую структуру. Внутри нее классы логически сгруппированы в так называемые *пространства имен*, которые служат для упорядочивания имен классов и предотвращения конфликтов имен: в разных пространствах имена могут совпадать. Пространства имен могут быть вложенными, их идея аналогична знакомой вам иерархической структуре каталогов на компьютере.

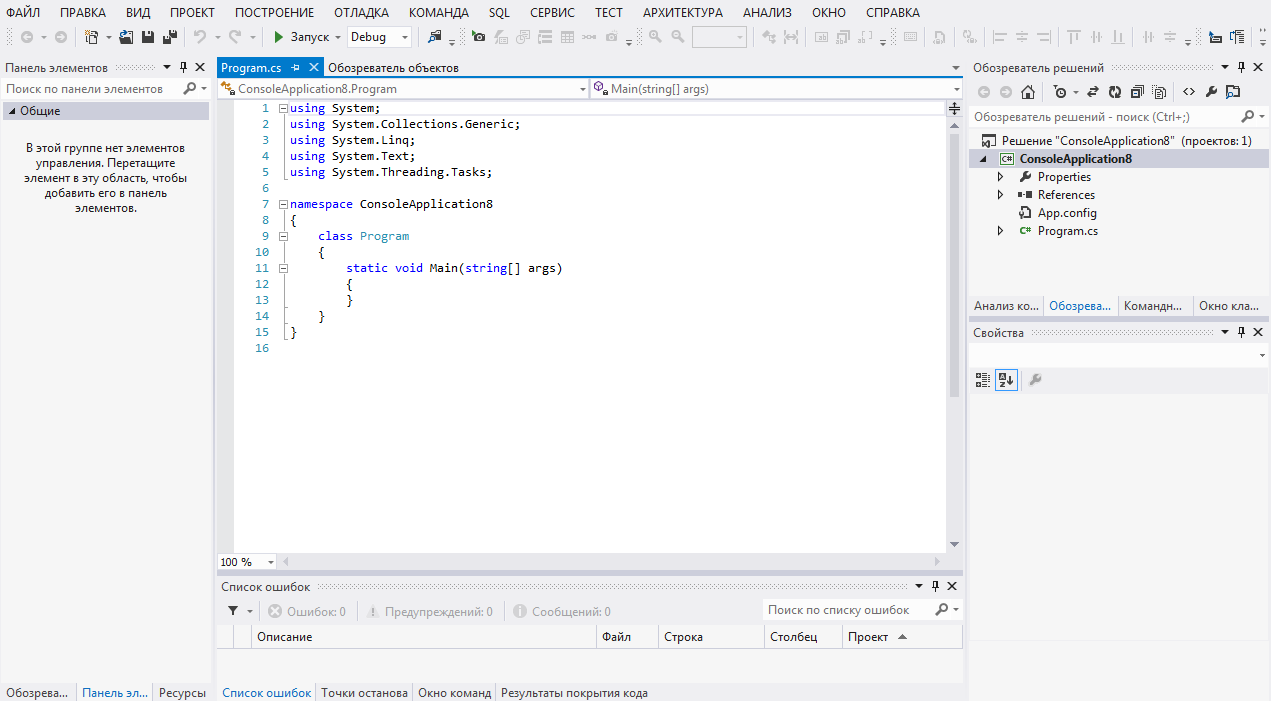
Любая программа, создаваемая в .NET, использует пространство имен System. В нем определены классы, которые обеспечивают базовую функциональность, например, поддерживают выполнение математических операций, управление памятью и ввод-вывод.

Обычно в одно пространство имен объединяют взаимосвязанные классы. Например,пространство System.Net содержит классы, относящиеся к передаче данных по сети, System.Windows.Forms — элементы графического интерфейса пользователя, такие как формы, кнопки и т. д. Имя каждого пространства имен представляет собой неделимую сущность, однозначно его определяющую.

Последнее, о чем необходимо поговорить, прежде чем начать последовательное изучение языка C#, — как создать простейшее приложение в среде Visual Studio.NET. Для того чтобы изучать именно язык программирования, а не сопутствующую информацию, мы будем работать с консольными приложениями. При запуске консольного приложения операционная система создает так называемое *консольное окно*, через которое идет весь ввод-вывод программы. Внешне это напоминает работу в операционной системе в режиме *командной строки*, когда ввод-вывод представляет собой поток символов.

## Простейшие приемы работы в среде

Для создания проекта следует после запуска Visual Studio.NET в главном меню выбрать команду **File** http://www.intuit.ru/img/tex/cb03bcd01df3f76828a6a116d40fc984.png **New** http://www.intuit.ru/img/tex/cb03bcd01df3f76828a6a116d40fc984.png **Project…**. В левой части открывшегося диалогового окна нужно выбрать пункт **Visual** **C#** **Projects**, в правой — пункт **Console** **Application**. В поле **Name** можно ввести имя проекта, а в поле **Location** — место его сохранения на диске, если заданные по умолчанию значения вас не устраивают. После щелчка на кнопке **OK** среда создаст решение и проект с указанным именем. Примерный вид экрана приведен на [рис. 1.1](http://www.intuit.ru/department/pl/phlcsharp/1/2.html#image.1.1).



**Рис. 1.1.** Примерный вид экрана после создания проекта консольного приложения

В верхней части экрана располагается *главное меню* (с разделами **File**, **Edit**, **View** и т. д.) и *панели инструментов* (toolbars). В верхней левой части экрана располагается *окно управления проектом* **Solution Explorer** (если оно не отображается, следует воспользоваться командой **View** http://www.intuit.ru/img/tex/cb03bcd01df3f76828a6a116d40fc984.png **Solution** **Explorer** главного меню). В окне перечислены все ресурсы, входящие в проект.

В этом же окне можно увидеть и другую информацию, если перейти на вкладку **Class View**, ярлычок которой находится в нижней части окна. На вкладке **Class View** представлен список всех классов, входящих в приложение, их элементов и предков.

В нижней левой части экрана расположено *окно свойств* **Properties** (если окна не видно, воспользуйтесь командой **View** http://www.intuit.ru/img/tex/cb03bcd01df3f76828a6a116d40fc984.png **Properties** главного меню). В окне свойств отображаются важнейшие характеристики выделенного элемента. Например, чтобы изменить имя файла, в котором хранится класс Class1, надо выделить этот файл в окне управления проектом и задать в окне свойств новое значение свойства **FileName** (ввод заканчивается нажатием клавиши **Enter** ).

Основное пространство экрана занимает *окно редактора*, в котором располагается текст программы, созданный средой автоматически. Текст представляет собой каркас, в который программист добавляет код по мере необходимости.

Слева от текста находятся *символы структуры*: щелкнув на любом квадратике с минусом, можно скрыть соответствующий блок кода. При этом минус превращается в плюс, щелкнув на котором можно опять вывести блок на экран. Это средство хорошо визуально структурирует код и позволяет сфокусировать внимание на нужных фрагментах.

#### Заготовка консольной программы

Рассмотрим каждую строку заготовки программы ([листинг 1.1](http://www.intuit.ru/department/pl/phlcsharp/1/2.html#example.1.1)). В зависимости от версии оболочки вид заготовки может немного варьироваться.

using System;

namespace ConsoleApplication1

{

/// <summary>

/// Summary description for Class1.

/// </summary>

class Class1

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main(string[] args)

{

//

// TODO: Add code to start application here

//

}

}

}

**Листинг 1.1. Заготовка консольной программы (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/1/example.1.1.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/1/example.1.1.txt)**)**

Директива using System разрешает использовать имена стандартных классов из пространства имен System непосредственно (без указания имени пространства).

Ключевое слово namespace создает для проекта собственное пространство имен, названное по умолчанию ConsoleApplication1. Это сделано для того, чтобы можно было давать программным объектам имена, не заботясь о том, что они могут совпасть с именами в других пространствах имен.

C# — объектно-ориентированный язык, поэтому написанная на нем программа представляет собой совокупность взаимодействующих между собой классов. В нашей заготовке программы всего один класс, которому по умолчанию задано имя Class1. *Описание класса* начинается с ключевого слова class, за которым следуют его имя и далее в фигурных скобках — список элементов класса (его данных и функций, называемых также методами).

**Внимание**

*Фигурные скобки являются важным элементом синтаксиса. Каждой открывающей скобке соответствует своя закрывающая, которая обычно располагается ниже по тексту с тем же отступом. Эти скобки ограничивают блок, внутри которого могут располагаться другие блоки, вложенные в него, как матрешки. Блок может применяться в любом месте, где допускается отдельный оператор*.

В данном случае внутри класса только один элемент — метод Main. Каждое приложение должно содержать метод Main — с него начинается выполнение программы. Все методы описываются по единым правилам.

*Упрощенный синтаксис метода*:

[ спецификаторы ] тип имя\_метода ( [ параметры ] )

{

тело метода: действия, выполняемые методом

}

Таким образом, любой метод должен иметь *тип*, *имя* и *тело*, остальные части описания являются необязательными. Среда заботливо поместила внутрь метода Mainкомментарий:

// TODO: Add code to start application here

Это означает: "Добавьте сюда код, выполняемый при запуске приложения".

Запуск консольного приложения лучше всего выполнять с помощью клавиш **Ctrl** + **F5** (или командой меню **Debug** http://www.intuit.ru/img/tex/cb03bcd01df3f76828a6a116d40fc984.png **Start** **Without** **Debugging** ). Компилятор может обнаружить в тексте программы *синтаксические ошибки*. Он сообщает об этом в окне, расположенном в нижней части экрана.

Теперь, когда вы получили общее представление о платформе .NET и основных понятиях ООП, можно приступить к планомерному изучению языка C#.

## Состав языка

Язык программирования можно уподобить примитивному иностранному языку с жесткими правилами без исключений. Изучение иностранного языка обычно начинают с алфавита, затем переходят к словам и законам построения фраз, и только в результате длительной практики и накопления *словарного запаса* появляется возможность свободно выражать на этом языке свои мысли. Примерно так же поступим и мы при изучении языка C#.

### Алфавит и лексемы

Все тексты на языке пишутся с помощью его *алфавита*. В C# используется кодировка символов Unicode. *Кодировкой*, или *кодовой таблицей* (character set), называется соответствие между символами и кодирующими их числами. Кодировка Unicode позволяет представить символы всех существующих алфавитов одновременно. Каждому символу соответствует свой уникальный код.

Алфавит C# включает:

* *буквы* (латинские и национальных алфавитов) и *символ подчеркивания* ( \_ ), который употребляется наряду с буквами;
* *цифры* ;
* *специальные символы*, например +, \*, { и & ;
* *пробельные символы* (пробел и символы табуляции);
* *символы перевода строки*.

Из символов составляются более крупные строительные блоки: лексемы, директивы препроцессора и комментарии.

*Лексема* (token) — это минимальная единица языка, имеющая самостоятельный смысл. Существуют следующие виды лексем:

* *имена* ( *идентификаторы* );
* *ключевые слова* ;
* *знаки операций* ;
* *разделители* ;
* *литералы* ( *константы* ).

Лексемы языка программирования аналогичны словам естественного языка. Например, лексемами являются число 128 (но не его часть 12), имя Vasia, ключевое слово goto и знак операции сложения +. Далее мы рассмотрим лексемы подробнее.

*Директивы препроцессора* пришли в C# из его предшественника — языка С++. Препроцессором называется предварительная стадия компиляции, на которой формируется окончательный вид исходного текста программы. Например, с помощью директив (инструкций, команд) препроцессора можно включить или выключить из процесса компиляции фрагменты кода. Директивы препроцессора не играют в C# такой важной роли, как в С++.

*Комментарии* предназначены для записи пояснений к программе и формирования документации. Правила записи комментариев мы рассмотрим чуть позже.

Из лексем составляются выражения и операторы. *Выражение* задает правило вычисления некоторого значения. Например, выражение a + b задает правило вычисления суммы двух величин.

*Оператор* задает законченное описание некоторого действия, данных или элемента программы. Например:

int a;

Это — оператор описания целочисленной переменной a.

### Идентификаторы

*Имена*, или *идентификаторы*, служат для того чтобы обращаться к программным объектам и различать их, то есть идентифицировать. В идентификаторе могут использоваться буквы, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например, *hacker*, *Hacker* и *hAcKeR* — три разных имени.

Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Длина идентификатора не ограничена. Пробелы внутри имен не допускаются.

В идентификаторах C# разрешается использовать, помимо латинских букв, буквы национальных алфавитов. Например, правильными являются идентификаторыФёкла и calc. Более того, можно применять даже так называемые *escape-последовательности Unicode*, то есть представлять символ с помощью его кода в шестнадцатеричном виде с префиксом \u, например, \u00F2.

**Примечание**

*Последняя возможность приведена здесь для полноты картины; не знаю, как вам, а мне трудно себе представить, зачем может понадобиться вставлять в имя шестнадцатеричные коды символов. По современным правилам хорошего стиля программирования имя обязано быть ясным, легко воспринимаемым и при этом как можно более точно отражать смысл и назначение именуемой величины*.

Имена даются элементам программы, к которым требуется обращаться: переменным, типам, константам, методам, меткам и т. д. Идентификатор создается на этапе объявления переменной (метода, типа и т. п.), после этого его можно использовать в последующих операторах программы. При выборе идентификатора необходимо следить, чтобы он не совпадал с ключевыми словами.

### Ключевые слова

*Ключевые слова* — это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Список ключевых слов C# приведен в [таблице 2.1](http://www.intuit.ru/department/pl/phlcsharp/2/#table.2.1). Как видите, их не так уж и много!

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 2.1. Ключевые слова C# | | | | |
| abstract | as | base | bool | break |
| byte | case | catch | char | checked |
| class | const | continue | decimal | default |
| delegate | do | double | else | enum |
| event | explicit | extern | false | finally |
| fixed | float | for | foreach | goto |
| if | implicit | in | int | interface |
| internal | is | lock | long | namespace |
| new | null | object | operator | out |
| override | params | private | protected | public |
| readonly | ref | return | sbyte | sealed |
| short | sizeof | stackalloc | static | string |
| struct | switch | this | throw | true |
| try | typeof | uint | ulong | unchecked |
| *unsafe* | ushort | using | virtual | void |
| volatile | while |  |  |  |

### Знаки операций и разделители

*Знак операции* — это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Например, в выражении a += b знак += является знаком операции, а a и b — операндами. Символы, составляющие знак операций, могут быть специальными, например, +, &&, |и <, и буквенными, такими как as или new.

Операции делятся на *унарные*, *бинарные* и *тернарную* по количеству участвующих в них операндов (один, два и три операнда соответственно). Один и тот же знак может интерпретироваться по-разному в зависимости от контекста.

*Разделители* используются для разделения или, наоборот, группирования элементов. Примеры разделителей: скобки, точка, запятая. Ниже перечислены все знаки операций и разделители, использующиеся в C#:

{ } [ ] ( ) . , : ; + - \* / % & | ^ ! ~ =

< > ? ++ -- && || << >> == != <= >= += -= \*= /= %=

&= |= ^= <<= >>= ->

### Литералы (константы)

*Литералами*, или *константами*, называют неизменяемые величины. В C# есть логические, целые, вещественные, символьные и строковые константы, а также константа null. Компилятор, выделив константу в качестве лексемы, относит ее к одному из типов данных по ее внешнему виду. Программист может задать тип константы и самостоятельно.

Описание и примеры констант каждого типа приведены в [таблице 2.2](http://www.intuit.ru/department/pl/phlcsharp/2/#table.2.2). Примеры, иллюстрирующие наиболее часто употребляемые формы констант, выделены полужирным шрифтом.

|  |  |  |
| --- | --- | --- |
| Таблица 2.2. Константы в C# | | |
| Константа | Описание | Примеры |
| Логическая | true (истина) или false (ложь) | true  false |
| Целая | *Десятичная*: последовательность десятичных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), за которой может следовать суффикс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu) | 8 0 199226  8u 0Lu 199226L |
|  | *Шестнадцатеричная*: символы 0х или 0Х, за которыми следуют шестнадцатеричные цифры (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), а за цифрами, в свою очередь, может следовать суффикс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu) | 0xA 0x1B8 0X00FF  0xAU 0x1B8LU  0X00FFl |
| Вещественная | *С фиксированной точкой*: [цифры][.][цифры][суффикс] Суффикс — один из символов F, f, D, d, M, m | 5.7 .001 35.  5.7F .001d 35.  5F .001f 35m |
|  | *С порядком*: [цифры][.][цифры]{E|e}[+|–][цифры] [суффикс] Суффикс — один из символов F, f, D, d, M, m | 0.2E6 .11e+3 5E-10  0.2E6D .11e–3  5E10 |
| Символьная | Символ, заключенный в апострофы | 'A' 'ю' '\*'  '\0' '\n'  '\xF' '\x74'  '\uA81B' |
| Строковая | Последовательность символов, заключенная в кавычки | "Здесь был Vasia"  "\tЗначение r = \xF5 \n"  "Здесь был \u0056\u0061"  "C:\\temp\\file1.txt"  @"C:\temp\file1.txt" |
| Константаnull | Ссылка, которая не указывает ни на какой объект | null |

*Логических* литералов всего два: true и false. Они широко используются в качестве признаков наличия или отсутствия чего-либо.

*Целые литералы* могут быть представлены либо в десятичной, либо в шестнадцатеричной системе счисления, а *вещественные* — только в десятичной системе, но в двух формах: с фиксированной точкой и с порядком. Вещественная константа с порядком представляется в виде *мантиссы* и *порядка*. Мантисса записывается слева от знака экспоненты ( E или e ), порядок — справа от знака. Значение константы определяется как произведение мантиссы и возведенного в указанную в порядке степень числа 10 (например, 1.3e2 = 1,3 x 102 = 130). Пробелы внутри константы не допускаются.

Если требуется сформировать *отрицательную* целую или вещественную константу, то перед ней ставится знак унарной операции изменения знака ( – ), например, –218.

*Символьная константа* представляет собой любой символ в кодировке Unicode. Символьные константы записываются в одной из четырех форм:

* "обычный" символ, имеющий графическое представление (кроме апострофа и символа перевода строки) — 'A', 'ю', '\*' ;
* управляющая последовательность — '\0', '\n' ;
* символ в виде шестнадцатеричного кода — '\xF', '\x74' ;
* символ в виде escape-последовательности Unicode — '\uA81B'.

*Управляющей последовательностью*, или *простой escape-последовательностью*, называют определенный символ, предваряемый обратной косой чертой. Управляющая последовательность интерпретируется как одиночный символ и используется для представления:

* кодов, не имеющих графического изображения (например, \n — переход в начало следующей строки);
* символов, имеющих специальное значение в строковых и символьных литералах, например, апострофа ( ' ).

Допустимые значения последовательностей приведены в [таблице 2.3](http://www.intuit.ru/department/pl/phlcsharp/2/#table.2.3).

|  |  |
| --- | --- |
| **Таблица 2.3. Управляющие последовательности в С#** | |
| **Вид** | **Наименование** |
| \a | Звуковой сигнал |
| \b | Возврат на шаг |
| \f | Перевод страницы (формата) |
| \n | Перевод строки |
| \r | Возврат каретки |
| \t | Горизонтальная табуляция |
| \v | Вертикальная табуляция |
| \\ | Обратная косая черта |
| \' | Апостроф |
| \" | Кавычка |
| \0 | *Нуль-символ* |

Символ, представленный в виде *шестнадцатеричного кода*, начинается с префикса \x, за которым следует код символа. Числовое значение должно находиться в диапазоне от 0 до 216 – 1, иначе возникает ошибка компиляции.

*Escape-последовательности Unicode* служат для представления символа в кодировке Unicode с помощью его кода в шестнадцатеричном виде с префиксом \u или \U, например, \u00F2, \U00010011. Коды в диапазоне от \U10000 до \U10FFFF представляются в виде двух последовательных символов; коды, превышающие \U10FFFF, не поддерживаются.

Управляющие последовательности обоих видов могут использоваться и в *строковых константах*, называемых иначе *строковыми литералами*. Например, если требуется вывести несколько строк, можно объединить их в один литерал, отделив одну строку от другой символами \n:

"Никто не доволен своей\nвнешностью, но каждый доволен\nсвоим умом"

Этот литерал при выводе будет выглядеть так:

Никто не доволен своей

внешностью, но каждый доволен

своим умом

В C# есть и второй вид литералов — *дословные* (verbatim strings). Эти литералы предваряются символом @, который отключает обработку управляющих последовательностей и позволяет получать строки в том виде, в котором они записаны. Чаще всего дословные литералы применяются при задании полного пути файла. Посмотрите, насколько лучше воспринимается второй вариант записи одного и того же пути:

"C:\\app\\bin\\debug\\a.exe"

@"C:\app\bin\debug\a.exe"

Строка может быть пустой (записывается парой смежных двойных кавычек "" ). Пустая символьная константа недопустима.

Константа null представляет собой значение, задаваемое по умолчанию для величин так называемых ссылочных типов, которые мы рассмотрим далее в этой лекции.

### Комментарии

*Комментарии* предназначены для записи пояснений к программе и формирования документации. Компилятор комментарии игнорирует. Внутри комментария можно использовать любые символы. В C# есть два вида комментариев: однострочные и многострочные.

*Однострочный* комментарий начинается с двух символов прямой косой черты ( // ) и заканчивается символом перехода на новую строку, *многострочный*заключается между символами-скобками /\* и \*/ и может занимать часть строки, целую строку или несколько строк. Комментарии не вкладываются друг в друга.

Кроме того, в языке есть еще одна разновидность комментариев, которые начинаются с трех подряд идущих символов косой черты ( /// ). Они предназначены для формирования документации к программе в формате XML. Компилятор извлекает эти комментарии из программы, проверяет их соответствие правилам и записывает их в отдельный файл.

### Типы данных

Данные, с которыми работает программа, хранятся в оперативной памяти. Естественно, что компилятору необходимо точно знать, сколько места они занимают, как именно закодированы и какие действия с ними можно выполнять. Все это задается при описании данных с помощью типа.

*Тип данных* однозначно определяет:

* *внутреннее представление* данных, а следовательно и *множество их возможных значений* ;
* *допустимые действия* над данными (операции и функции).

Например, целые и вещественные числа, даже если они занимают одинаковый объем памяти, имеют совершенно разные диапазоны возможных значений.

Каждое выражение в программе имеет определенный тип. Компилятор использует информацию о типе при проверке допустимости описанных в программе действий.

Память, в которой хранятся данные во время выполнения программы, делится на две области: стек (stack) и динамическая область, или хип (heap), чаще называемый кучей (поскольку этот термин мне не нравится, я его использовать не буду). *Стек*используется для хранения величин, память под которые выделяет компилятор, а в *динамической области* память резервируется и освобождается во время выполнения программы с помощью специальных команд. Основным местом для хранения данных в C# является хип.

### Классификация типов

Типы можно классифицировать по разным признакам. Если принять за основу строение элемента, все типы можно разделить на *простые* (не имеют внутренней структуры) и *структурированные* (состоят из элементов других типов). По своему "создателю" типы можно разделить на*встроенные* (стандартные) и *определяемые программистом*. По способу хранения значений типы делятся на *значимые*, или типы-значения, и *ссылочные*. Рассмотрим в первую очередь встроенные типы C#.

### Встроенные типы

Встроенные типы не требуют предварительного определения. Для каждого типа существует ключевое слово, которое используется при описании переменных, констант и т. д. Встроенные типы C# приведены в [таблице 2.4](http://www.intuit.ru/department/pl/phlcsharp/2/2.html#table.2.4). Они однозначно соответствуют стандартным классам библиотеки .NET, определенным в пространстве имен System. Как видно из таблицы, существуют несколько вариантов представления целых и вещественных величин. Программист выбирает тип каждой величины, используемой в программе, с учетом необходимого ему диапазона и точности представления данных.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 2.4. Встроенные типы C# | | | | | |
| Название | Ключевое слово | Тип .NET | Диапазон значений | Описание | Размер, битов |
| Логический тип | bool | Boolean | true, false |  |  |
| Целые типы | sbyte | SByte | От –128 до 127 | Со знаком | 8 |
| byte | Byte | От 0 до 255 | Без знака | 8 |
| short | Int16 | От –32768 до 32767 | Со знаком | 16 |
| ushort | UInt16 | От 0 до 65535 | Без знака | 16 |
| int | Int32 | От –2 x 109 до 2 x 109 | Со знаком | 32 |
| uint | UInt32 | От 0 до 4 x 109 | Без знака | 32 |
| long | Int64 | От –9 x 1018 до 9 x 1018 | Со знаком | 64 |
| ulong | UInt64 | От 0 до 18 x 1018 | Без знака | 64 |
| Символьный тип | char | Char | От U+0000 до U+ffff | Unicode-символ | 16 |
| Вещественные | float | Single | От 1.5 x 10-45 до 3.4 x 1038 | 7 цифр | 32 |
| double | Double | От 5.0 x 10-324 до 1.7 x 10308 | 15–16 цифр | 64 |
| Финансовый тип | decimal | Decimal | От 1.0 x 10-28 до 7.9 x 1028 | 28–29 цифр | 128 |
| Строковый тип | string | String | Длина ограничена объемом доступной памяти | Строка из Unicode-символов |  |
| Тип object | object | Object | Можно хранить все, что угодно | Всеобщий предок |  |

**Примечание**

*Вещественные типы и финансовый тип являются знаковыми. Для них в столбце "диапазон значений" приведены абсолютные величины допустимых значений*.

*Логический*, или булев, тип содержит всего два значения: true (истина) и false (ложь). Их внутреннее представление программиста интересовать не должно. Все целые и вещественные типы вместе с символьным и финансовым можно назвать *арифметическими типами*.

Внутреннее представление величины *целого типа* — целое число в двоичном коде. В знаковых типах старший бит числа интерпретируется как знаковый (0 — положительное число, 1 — отрицательное).

*Вещественные типы*, или *типы данных с плавающей точкой*, хранятся в памяти компьютера иначе, чем целочисленные. Внутреннее представление вещественного числа состоит из двух частей — мантиссы и порядка, причем каждая часть имеет знак. Длина мантиссы определяет *точность* числа, а длина порядка — его*диапазон*. Все вещественные типы могут представлять как положительные, так и отрицательные числа. Чаще всего в программах используется тип double, поскольку его диапазон и точность покрывают большинство потребностей. Этот тип имеют вещественные литералы и многие стандартные математические функции.

Тип decimal предназначен для *денежных вычислений*, в которых критичны ошибки округления. Величины типа decimal позволяют хранить 28–29 десятичных разрядов. Тип decimal не относится к вещественным типам, у них различное внутреннее представление. Величины *денежного типа* даже нельзя использовать в одном выражении с вещественными без явного преобразования типа. Использование величин финансового типа в одном выражении с целыми допускается.

Любой встроенный тип C# соответствует стандартному классу библиотеки .NET, определенному в пространстве имен System. Имя этого класса приведено в третьем столбце таблицы 2.4. Везде, где используется имя встроенного типа, его можно заменить именем класса библиотеки.

### Типы литералов

Литералы (константы) тоже имеют тип. Если значение целого литерала находится внутри диапазона допустимых значений типа int, литерал рассматривается как int, иначе он относится к наименьшему из типов uint, long или ulong, в диапазон значений которого он входит. Вещественные литералы по умолчанию относятся к типу double.

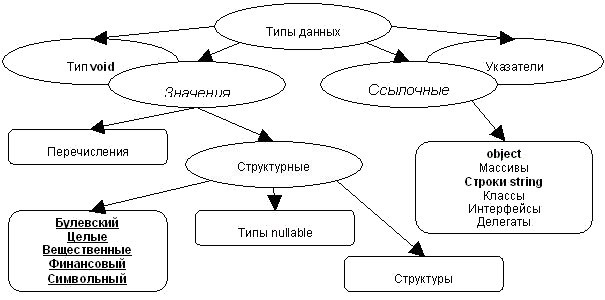
Например, константа 10 относится к типу int, хотя для ее хранения достаточно и байта, а константа 2147 483 648 будет определена как uint. Для явного задания типа литерала служит суффикс, например, 1.1f, 1UL, 1000m. Явное задание применяется в основном для уменьшения количества *неявных преобразований типа*, выполняемых компилятором.

### Типы-значения и ссылочные типы

Чаще всего типы C# разделяют *по способу хранения элементов* на типы-значения и ссылочные типы ([рис. 2.1](http://www.intuit.ru/department/pl/phlcsharp/2/2.html#image.2.1)). Элементы *типов-значений*, или *значимых типов* (value types), представляют собой просто последовательность битов в памяти, необходимый объем которой выделяет компилятор. Иными словами, величины значимых типов хранят свои значения непосредственно. Величина *ссылочного типа* хранит не сами данные, а ссылку на них (адрес, по которому расположены данные). Сами данные хранятся в хипе.

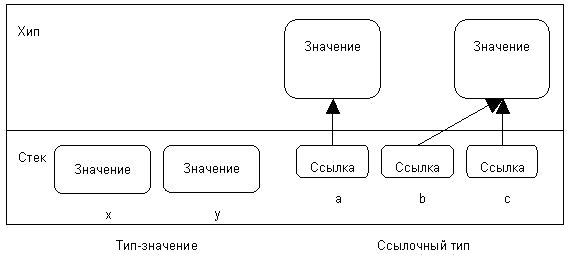
Внимание

*Несмотря на различия в способе хранения, и типы-значения, и ссылочные типы являются потомками общего базового класса* object.



**Рис. 2.1.** Классификация типов данных C# по способу хранения

[рис. 2.2](http://www.intuit.ru/department/pl/phlcsharp/2/2.html#image.2.2) иллюстрирует разницу между величинами значимого и ссылочного типов. Одни и те же действия над ними выполняются по-разному. Рассмотрим в качестве примера проверку на равенство. Величины значимого типа равны, если равны их значения. Величины ссылочного типа равны, если они ссылаются на одни и те же данные (на рисунке b и c равны, но a не равно b даже при одинаковых значениях). Из этого следует, что если изменить значение одной величины ссылочного типа, это может отразиться на другой.



**Рис. 2.2.** Хранение в памяти величин значимого и ссылочного типов

### Упаковка и распаковка

Для того чтобы величины ссылочного и значимого типов могли использоваться совместно, необходимо иметь возможность преобразования из одного типа в другой. Преобразование из типа-значения в ссылочный тип называется *упаковкой* (boxing), обратное преобразование — *распаковкой* (unboxing).

Если величина значимого типа используется в том месте, где требуется ссылочный тип, автоматически выполняется создание промежуточной величины ссылочного типа. При необходимости обратного преобразования с величины ссылочного типа "снимается упаковка", и в дальнейших действиях участвует только ее значение.

# Переменные, операции, выражения

*Переменная* — это *именованная область* памяти, предназначенная для хранения данных определенного типа. Во время выполнения программы значение переменной можно изменять. Все переменные, используемые в программе, должны быть описаны явным образом. При описании для каждой переменной задаются ее *имя* и *тип*.

Пример описания целой переменной с именем a и вещественной переменной x:

int a; float x;

*Имя* переменной служит для обращения к области памяти, в которой хранится*значение* переменной. Имя дает программист. *Тип переменной выбирается, исходя из диапазона и требуемой точности представления данных*. При объявлении можно присвоить переменной некоторое начальное значение, то есть *инициализировать* ее, например:

int a, b = 1;

float x = 0.1, y = 0.1f;

Здесь описаны:

* переменная a типа int, начальное значение которой не присваивается;
* переменная b типа int, ее начальное значение равно 1;
* переменные х и y типа float, которым присвоены одинаковые начальные значения 0.1. Разница между ними состоит в том, что для инициализации переменной х сначала формируется константа типа double, а затем она преобразуется к типу float ; переменной y значение 0.1 присваивается без промежуточного преобразования.

Рекомендуется всегда инициализировать переменные при описании. При инициализации можно использовать не только константу, но и выражение — главное, чтобы на момент описания оно было вычисляемым, например:

int b = 1, a = 100;

int x = b \* a + 25;

Программа на C# состоит из классов, внутри которых описывают методы и данные. Структуру программы иллюстрирует [рис. 3.1](http://www.intuit.ru/department/pl/phlcsharp/3/#image.3.1). Переменные, описанные непосредственно внутри класса, называются *полями класса*. Им автоматически присваивается так называемое "значение по умолчанию" — как правило, это 0 соответствующего типа. Переменные, описанные внутри метода класса, называются *локальными переменными*. Их инициализация возлагается на программиста.



**Рис. 3.1.** Структура программы

Так называемая *область действия* переменной, то есть область программы, где можно использовать переменную, начинается в точке ее описания и длится до конца блока, внутри которого она описана. *Блок* — это код, заключенный в фигурные скобки. Основное назначение блока — группировка операторов. В C# любая переменная описана внутри какого-либо блока: класса, метода или блока внутри метода. Имя переменной должно быть уникальным в области ее действия. Область действия распространяется на вложенные в метод блоки.

class X // начало описания класса X

{

int A; // поле A класса X

int B; // поле B класса X

void Y() // ----------------------------------------- метод Y класса Х

{

int C; // локальная переменная C, область действия – метод Y

int A; // локальная переменная A (НЕ конфликтует с полем А)

{ // ============ вложенный блок 1 ============

int D; // локальная переменная D, область действия – этот блок

// int А; недопустимо! Ошибка компиляции, конфликт с локальной

// переменной А

С = B; // присваивание переменной С поля В класса Х (\*\*)

С = this.A; // присваивание переменной С поля А класса Х (\*\*\*)

} // ============ конец блока 1 ===============

{ // ============ вложенный блок 2 ============

int D; // локальная переменная D, область действия – этот блок

} // ============ конец блока 2 ===============

} // ------------------------- конец описания метода Y класса X

} // конец описания класса X

В [листинге 3.1](http://www.intuit.ru/department/pl/phlcsharp/3/#example.3.1) приведен пример программы, в которой описываются и выводятся на экран локальные переменные.

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

int i = 3;

double y = 4.12;

decimal d = 600m;

string s = "Вася";

Console.Write( "i = " ); Console.WriteLine( i );

Console.Write( "y = " ); Console.WriteLine( y );

Console.Write( "d = " ); Console.WriteLine( d );

Console.Write( "s = " ); Console.WriteLine( s );

}

}

}

**Листинг 3.1. Описание переменных (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/3/example.3.1.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/3/example.3.1.txt)**)**

Внимание

*Переменные создаются при входе в их область действия (блок) и уничтожаются при выходе. Это означает, что после выхода из блока значение переменной не сохраняется. При повторном входе в этот же блок переменная создается заново*.

### Именованные константы

Можно запретить изменять значение переменной, задав при ее описании ключевое слово const, например:

const int b = 1;

const float x = 0.1, y = 0.1f; // const распространяется на обе переменные

Такие величины называют *именованными константами*, или просто *константами*. Они применяются для того, чтобы вместо значений констант можно было использовать в программе их имена. Это делает программу более понятной и облегчает внесение в нее изменений. Именованные константы должны обязательно инициализироваться при описании, например:

const int b = 1, a = 100;

const int x = b \* a + 25;

### Операции и выражения

*Выражение* — это правило вычисления значения. В выражении участвуют *операнды*, объединенные *знаками операций*. Операндами простейшего выражения могут быть константы, переменные и вызовы функций.

Например, a + 2 — это выражение, в котором + является знаком операции, а a и 2 — операндами. Пробелы внутри знака операции, состоящей из нескольких символов, не допускаются. Операции C# приведены в [таблице 3.1](http://www.intuit.ru/department/pl/phlcsharp/3/#table.3.1).

|  |  |  |
| --- | --- | --- |
| Таблица 3.1. Операции C# | | |
| Категория | Знак операции | Название |
| Первичные | x() | Вызов метода или делегата |
| x[] | Доступ к элементу |
| x++ | Постфиксный инкремент |
| x-- | Постфиксный декремент |
| new | Выделение памяти |
| typeof | Получение типа |
| checked | Проверяемый код |
| unchecked | Непроверяемый код |
| Унарные | + | Унарный плюс |
| - | Унарный минус (арифметическое отрицание) |
| ! | Логическое отрицание |
| http://www.intuit.ru/img/tex/99da5a219f3778bb98c786a61570ae61.png | Поразрядное отрицание |
| ++x | Префиксный инкремент |
| --x | Префиксный декремент |
| (тип)x | Преобразование типа |
| Мультипликативные (типа умножения) | \* | Умножение |
| / | Деление |
| % | Остаток от деления |
| Аддитивные (типа сложения) | + | Сложение |
| - | Вычитание |
| Сдвига | << | Сдвиг влево |
| >> | Сдвиг вправо |
| Отношения и проверки типа | < | Меньше |
| > | Больше |
| <= | Меньше или равно |
| >= | Больше или равно |
| is | Проверка принадлежности типу |
| as | Приведение типа |
| Проверки на равенство | == | Равно |
| != | Не равно |
| Поразрядные логические | & | Поразрядная конъюнкция (И) |
| http://www.intuit.ru/img/tex/f7c27b287dbd7d0efefae4b048962bc6.png | Поразрядное исключающее ИЛИ |
| | | Поразрядная дизъюнкция (ИЛИ) |
| Условные логические | && | Логическое И |
| || | Логическое ИЛИ |
| Условная | ?: | Условная операция |
| Присваивания | = | Присваивание |
| \*= | Умножение с присваиванием |
| /= | Деление с присваиванием |
| %= | Остаток отделения с присваиванием |
| += | Сложение с присваиванием |
| -= | Вычитание с присваиванием |
| <<= | Сдвиг влево с присваиванием |
| >>= | Сдвиг вправо с присваиванием |
| &= | Поразрядное И с присваиванием |
| http://www.intuit.ru/img/tex/2d2f863d6fdfe19256303dc9b6507584.png | Поразрядное исключающее ИЛИ с присваиванием |
| |= | Поразрядное ИЛИ с присваиванием |

Операции в выражении выполняются в определенном порядке в соответствии с *приоритетами*, как и в математике. В [таблице 3.1](http://www.intuit.ru/department/pl/phlcsharp/3/#table.3.1) операции расположены по убыванию приоритетов, уровни приоритетов разделены в таблице горизонтальными линиями.

Результат вычисления выражения характеризуется *значением* и *типом*. Например, пусть a и b — переменные целого типа и описаны так:

int a = 2, b = 5;

Тогда выражение a + b имеет значение 7 и тип int, а выражение a = b имеет значение, равное помещенному в переменную a (в данном случае — 5) и тип, совпадающий с типом этой переменной.

Если в одном выражении соседствует несколько операций одинакового приоритета, операции присваивания и условная операция выполняются *справа налево*, остальные — *слева направо*. Для изменения порядка выполнения операций используются *круглые* *скобки*, уровень их вложенности практически не ограничен.

Например, a + b + c означает (a + b) + c, а a = b = c означает a = (b = c).

### Преобразования встроенных арифметических типов-значений

При вычислении выражений может возникнуть необходимость в преобразовании типов. *Если операнды*, *входящие в выражение*, *одного типа*, *и операция для этого типа определена*, *то результат выражения будет иметь тот же тип*.

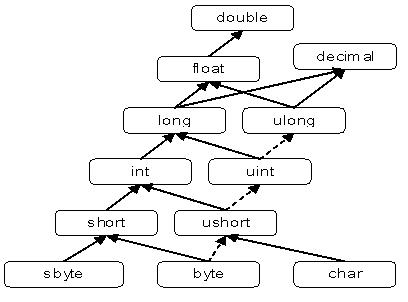
Если операнды разного типа и (или) операция для этого типа не определена, перед вычислениями автоматически выполняется преобразование типа по правилам, обеспечивающим приведение более коротких типов к более длинным для сохранения значимости и точности. Автоматическое ( *неявное* ) преобразование возможно не всегда, а только если при этом не может случиться потеря значимости.

Если неявного преобразования из одного типа в другой не существует, программист может задать *явное* преобразование типа с помощью операции (тип)x. Его результат остается на совести программиста.

**Внимание**

Арифметические операции не определены для более коротких, чем int, типов. Это означает, что если в выражении участвуют только величины типов sbyte, byte,short и ushort, перед выполнением операции они будут преобразованы в int. Таким образом, результат любой арифметической операции имеет тип не менее int.

Правила неявного преобразования иллюстрирует [рис. 3.2](http://www.intuit.ru/department/pl/phlcsharp/3/#image.3.2). Если один из операндов имеет тип, изображенный на более низком уровне, чем другой, то он приводится к типу второго операнда при наличии пути между ними. Если пути нет, возникает ошибка компиляции.



**Рис. 3.2.** Неявные арифметические преобразования типов

### Введение в исключения

При вычислении выражений могут возникнуть ошибки, например, переполнение, *исчезновение порядка* или деление на ноль. В C# есть механизм, который позволяет обрабатывать подобные ошибки и таким образом избегать аварийного завершения программы. Он так и называется: *механизм обработки исключительных ситуаций* ( *исключений* ).

Если в процессе вычислений возникла ошибка, система сигнализирует об этом с помощью специального действия, называемого *выбрасыванием (генерированием) исключения*. Каждому типу ошибки соответствует свое исключение. Поскольку C# — язык объектно-ориентированный, исключения являются классами, которые имеют общего предка — класс Exception, определенный в пространстве имен System.

Например, при делении на ноль будет сгенерировано исключение DivideByZeroException, при недостатке памяти — исключение OutOfMemoryException.

Программист может задать способ обработки исключения в специальном блоке кода, начинающемся с ключевого слова catch ("перехватить"), который будет автоматически выполнен при возникновении соответствующей исключительной ситуации. Внутри блока можно, например, вывести предупреждающее сообщение или скорректировать значения величин и продолжить выполнение программы. Если этот блок не задан, система выполнит *действия по умолчанию*, которые обычно заключаются в выводе диагностического сообщения и нормальном завершении программы. Обработка исключений подробно рассматривается позже.

# Справочная информация: Основные операции C#

В этом разделе кратко описаны синтаксис и применение всех операций C#, кроме некоторых первичных, которые рассматриваются позже при изучении соответствующего материала.

### Инкремент и декремент

Операции инкремента (++) и декремента (--) увеличивают и уменьшают операнд на единицу. Они имеют две формы записи —*префиксную*, когда знак операции записывается перед операндом, и *постфиксную*. В префиксной форме сначала изменяется операнд, а затем его значение становится результирующим значением выражения, а в постфиксной форме значением выражения является исходное значение операнда, после чего он изменяется.

Стандартные операции инкремента существуют для целых, символьных, вещественных и финансовых величин.

### Операция new

Операция new служит для создания нового объекта. Формат операции:

new тип ( [ аргументы ] )

С помощью этой операции можно создавать объекты как ссылочных, так и значимых типов, например:

object z = new object();

int i = new int(); // то же самое, что int i = 0;

При выполнении операции new сначала выделяется необходимый объем памяти (для ссылочных типов в хипе, для значимых — в стеке), а затем вызывается так называемый *конструктор по умолчанию*, то есть метод, с помощью которого инициализируется объект. Переменной *значимого типа* присваивается *значение по умолчанию*, которое равно нулю соответствующего типа.

### Операции отрицания

*Арифметическое отрицание* (унарный минус – ) меняет знак операнда на противоположный. Стандартная операция отрицания определена для типов int, long,float, double и decimal. К величинам других типов ее можно применять, если для них возможно неявное преобразование к этим типам.

*Логическое* *отрицание* ( !) определено для типа bool. Результат операции — значение false, если операнд равен true, и значение true, если операнд равен false.

*Поразрядное отрицание* ( ~ ), часто называемое побитовым, инвертирует каждый разряд в двоичном представлении операнда типа int, uint, long или ulong.

### Явное преобразование типа

Операция используется для явного преобразования величины из одного типа в другой. Это требуется в том случае, когда неявного преобразования не существует. При преобразовании из более длинного типа в более короткий возможна потеря информации. Формат операции:

( тип ) выражение

Здесь тип — это имя того типа, в который осуществляется преобразование, а выражение чаще всего представляет собой имя переменной, например:

long b = 300;

int a = (int) b; // данные не теряются

byte d = (byte) a; // данные теряются

### Умножение, деление и остаток от деления

*Операция умножения* ( \* ) возвращает результат перемножения двух операндов. Стандартная операция умножения определена для типов int, uint, long, ulong,float, double и decimal. К величинам других типов ее можно применять, если для них возможно неявное преобразование к этим типам. Тип результата операции равен "наибольшему" из типов операндов, но не менее int.

Все возможные значения для вещественных операндов приведены в [таблице 3.2](http://www.intuit.ru/department/pl/phlcsharp/3/2.html#table.3.2). Символами х и y обозначены конечные положительные значения, символом z — результат операции вещественного умножения. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению "бесконечность", если слишком мал, он принимается за 0. NaN (not a number) означает, что результат не является числом.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.2. Результаты вещественного умножения | | | | | | | |
| \* | +y | -y | +0 | -0 | +http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | -http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| +x | +z | -z | +0 | -0 | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| -x | -z | +z | -0 | +0 | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| +0 | +0 | -0 | +0 | -0 | NaN | NaN | NaN |
| -0 | -0 | +0 | -0 | +0 | NaN | NaN | NaN |
| +http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN | NaN | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| -http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN | NaN | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

*Операция деления* ( / ) вычисляет частное от деления первого операнда на второй. Стандартная операция деления определена для типов int, uint, long, ulong,float, double и decimal. К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам. Тип результата определяется правилами преобразования, но не меньше int.

Если оба операнда целочисленные, результат операции округляется вниз до ближайшего целого числа. Если делитель равен нулю, генерируется исключениеSystem.DivideByZeroException.

Если хотя бы один из операндов вещественный, дробная часть результата деления не отбрасывается, а все возможные значения приведены в [таблице 3.3](http://www.intuit.ru/department/pl/phlcsharp/3/2.html#table.3.3).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Таблица 3.3. Результаты вещественного деления** | | | | | | | |
| **/** | **+y** | **-y** | **+0** | **-0** | **+ http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png** | **- http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png** | **NaN** |
| **+x** | +z | -z | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | +0 | -0 | NaN |
| **-x** | -z | +z | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | -0 | +0 | NaN |
| **+0** | +0 | -0 | NaN | NaN | +0 | -0 | NaN |
| **-0** | -0 | +0 | NaN | NaN | -0 | +0 | NaN |
| **+ http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png** | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN | NaN | NaN |
| **- http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png** | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN | NaN | NaN |
| **NaN** | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

Для финансовых величин (тип decimal ) при делении на 0 и переполнении генерируются соответствующие исключения, при *исчезновении порядка* результат равен 0.

*Операция остатка от деления* ( % ) также интерпретируется по-разному для целых, вещественных и финансовых величин. Если оба операнда целочисленные, результат операции вычисляется по формуле x - (x / y) \* y. Если делитель равен нулю, генерируется исключение System.DivideByZeroException.

Если хотя бы один из операндов вещественный, результат операции вычисляется по формуле x – n \* y, где n — наибольшее целое, меньшее или равное результату деления х на y. Все возможные комбинации значений операндов приведены в [таблице 3.4](http://www.intuit.ru/department/pl/phlcsharp/3/2.html#table.3.4).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Таблица 3.4. Результаты вещественного остатка от деления** | | | | | | | |
| **%** | **+y** | **-y** | **+0** | **-0** | **+ http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png** | **- http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png** | **NaN** |
| **+x** | +z | z | NaN | NaN | x | x | NaN |
| **-x** | -z | -z | NaN | NaN | -x | -x | NaN |
| **+0** | +0 | +0 | NaN | NaN | +0 | +0 | NaN |
| **-0** | -0 | -0 | NaN | NaN | -0 | -0 | NaN |
| **+ http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png** | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **- http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png** | NaN | NaN | NaN | NaN | NaN | NaN | NaN |
| **NaN** | NaN | NaN | NaN | NaN | NaN | NaN | NaN |

Для финансовых величин (тип decimal ) при получении остатка от деления на 0 и при переполнении генерируются соответствующие исключения, при*исчезновении порядка* результат равен 0. Знак результата равен знаку первого операнда.

### Сложение и вычитание

*Операция сложения* ( + ) возвращает сумму двух операндов. Стандартная операция сложения определена для типов int, uint, long, ulong, float, double и decimal. К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам. Тип результата операции равен "наибольшему" из типов операндов, но не менее int.

Если оба операнда целочисленные или типа decimal и результат операции слишком велик для представления с помощью заданного типа, генерируется исключение System.OverflowException.

Все возможные значения для вещественных операндов приведены в [таблице 3.5](http://www.intuit.ru/department/pl/phlcsharp/3/2.html#table.3.5).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.5. Результаты вещественного сложения | | | | | | |
| + | y | +0 | -0 | +http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | -http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| x | z | x | x | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| +0 | y | +0 | +0 | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| -0 | y | +0 | -0 | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| +http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN | NaN |
| -http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

*Операция вычитания* ( - ) возвращает разность двух операндов. Стандартная операция вычитания определена для типов int, uint, long, ulong, float, double иdecimal. К величинам других типов ее можно применять, если для них существует неявное преобразование к этим типам. Тип результата операции равен "наибольшему" из типов операндов, но не менее int.

Если оба операнда целочисленные или типа decimal и результат операции слишком велик для представления с помощью заданного типа, генерируется исключение System.OverflowException.

Все возможные значения результата вычитания для вещественных операндов приведены в [таблице 3.6](http://www.intuit.ru/department/pl/phlcsharp/3/2.html#table.3.6). Символами х и y обозначены конечные положительные значения, символом z — результат операции вещественного вычитания. Если х и y равны, результат равен положительному нулю. Если результат слишком велик для представления с помощью заданного типа, он принимается равным значению "бесконечность" с тем же знаком, что х - y, если слишком мал, он принимается за 0 с тем же знаком, что х - y.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Таблица 3.6. Результаты вещественного вычитания | | | | | | |
| - | y | +0 | -0 | +http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | -http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| x | z | x | x | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| +0 | -y | +0 | +0 | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| -0 | -y | -0 | +0 | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| +http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN | + http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN |
| -http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | - http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png | NaN | NaN |
| NaN | NaN | NaN | NaN | NaN | NaN | NaN |

### Операции сдвига

*Операции сдвига* ( << и >> ) применяются к целочисленным операндам. Они сдвигают двоичное представление первого операнда влево или вправо на количество двоичных разрядов, заданное вторым операндом.

При *сдвиге влево* ( << ) освободившиеся разряды обнуляются. При *сдвиге вправо* ( >> ) освободившиеся биты заполняются нулями, если первый операнд беззнакового типа, и знаковым разрядом в противном случае. Стандартные операции сдвига определены для типов int, uint, long и ulong.

### Операции отношения и проверки на равенство

*Операции отношения* ( <, <=, >, >=, ==, != ) сравнивают первый операнд со вторым. Операнды должны быть арифметического типа. Результат операции — логического типа, равен true или false. Правила вычисления результатов приведены в [таблице 3.7](http://www.intuit.ru/department/pl/phlcsharp/3/2.html#table.3.7).

|  |  |
| --- | --- |
| Таблица 3.7. Результаты операций отношения | |
| Операция | Результат |
| x == y | true, если x равно y, иначе false |
| x != y | true, если x не равно y, иначе false |
| x < y | true, если x меньше y, иначе false |
| x > y | true, если x больше y, иначе false |
| x <= y | true, если x меньше или равно y, иначе false |
| x >= y | true, если x больше или равно y, иначе false |

### Поразрядные логические операции

*Поразрядные логические операции* ( &, |, ^ ) применяются к целочисленным операндам и работают с их двоичными представлениями. При выполнении операций операнды сопоставляются побитно (первый бит первого операнда с первым битом второго, второй бит первого операнда со вторым битом второго, и т д.). Стандартные операции определены для типов int, uint, long и ulong.

При *поразрядной конъюнкции* ( & ), бит результата равен 1 только тогда, когда соответствующие биты обоих операндов равны 1.

При *поразрядной дизъюнкции* ( | ), бит результата равен 1 тогда, когда соответствующий бит хотя бы одного из операндов равен 1.

*При поразрядном исключающем ИЛИ* ( http://www.intuit.ru/img/tex/f7c27b287dbd7d0efefae4b048962bc6.png ) бит результата равен 1 только тогда, когда соответствующий бит только одного из операндов равен 1.

### Условные логические операции

*Условные логические операции* И ( && ) и ИЛИ ( || ) чаще всего используются с операндами логического типа. Результатом логической операции является true илиfalse. Операции вычисляются по сокращенной схеме.

Результат операции *логическое И* имеет значение true, только если оба операнда имеют значение true. Результат операции *логическое ИЛИ* имеет значение true, если хотя бы один из операндов имеет значение true. Если значения первого операнда достаточно, чтобы определить результат операции, второй операнд не вычисляется.

### Условная операция

Условная операция ( ? :) имеет три операнда. Ее формат:

операнд\_1 ? операнд\_2 : операнд\_3

Первый операнд — выражение, для которого существует неявное преобразование к логическому типу. Если результат вычисления первого операнда равен true, то результатом условной операции будет значение второго операнда, иначе — третьего операнда. Вычисляется всегда либо второй операнд, либо третий. Их тип может различаться.

Тип результата операции зависит от типа второго и третьего операндов. Если операнды одного типа, он и становится типом результата операции.

### Операции присваивания

*Операции присваивания* ( =, +=, -=, \*= и т. д.) задают новое значение переменной. Эти операции могут использоваться в программе как законченные операторы.

Формат операции *простого присваивания* ( = ):

переменная = выражение

Механизм выполнения операции присваивания такой: вычисляется выражение и его результат заносится в память по адресу, который определяется именем переменной, находящейся слева от знака операции. То, что ранее хранилось в этой области памяти, теряется. Примеры операторов присваивания:

a = b + c / 2;

x = 1;

x = x + 0.5;

*Для правого операнда операции присваивания должно существовать неявное преобразование к типу левого операнда*. Например, выражение целого типа можно присвоить вещественной переменной, потому что целые числа являются подмножеством вещественных, и информация при таком присваивании не теряется.

вещественная\_переменная := целое\_выражение;

Результатом операции присваивания является значение, записанное в левый операнд. Тип результата совпадает с типом левого операнда.

*В сложных операциях присваивания* ( +=, \*=, /= и т п.) при вычислении выражения, стоящего в правой части, используется значение из левой части. Например, при*сложении с присваиванием* ко второму операнду прибавляется первый, и результат записывается в первый операнд, то есть выражение a += b является более компактной записью выражения a = a + b.

Результатом операции сложного присваивания является значение, записанное в левый операнд.

Операции присваивания *правоассоциативны*, то есть выполняются справа налево, в отличие от большинства других операций ( a = b = c означает a = (b = c) ).

# Простейший ввод-вывод. Управляющие операторы

### **Консольный** ввод-вывод

Любая программа при вводе исходных данных и выводе результатов взаимодействует с внешними устройствами. Совокупность стандартных устройств ввода и вывода, то есть клавиатуры и экрана, называется *консолью*. В языке C# нет операторов ввода и вывода. Вместо них для обмена с внешними устройствами применяются стандартные объекты. Для работы с консолью в C# применяется класс Console, определенный в пространстве имен System. Методы этого класса Write и WriteLine уже использовались в [листинге 3.1](http://www.intuit.ru/department/pl/phlcsharp/3/1.html#example.3.1). Поговорим о них подробнее, для чего внесем некоторые изменения в этот листинг. Результаты этих изменений представлены в [листинге 4.1](http://www.intuit.ru/department/pl/phlcsharp/4/#example.4.1).

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

int i = 3;

double y = 4.12;

decimal d = 600m;

string s = "Вася";

Console.WriteLine( "i = " + i ); // 1

Console.WriteLine( "s = " + s ); // 2

Console.WriteLine( "y = {0} \n d = {1}", y, d ); // 3

}

}

}

**Листинг 4.1. Методы вывода (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.1.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.1.txt)**)**

Результат работы программы:

i = 3

s = Вася

y = 4,12

d = 600

До сих пор мы использовали метод WriteLine для вывода значений переменных и литералов различных встроенных типов. Это возможно благодаря тому, что в классе Console существует несколько вариантов методов с именами Write и WriteLine, предназначенных для вывода значений различных типов.

Методы с одинаковыми именами, но разными параметрами называются *перегруженными*. Компилятор определяет, какой из методов вызван, по типу передаваемых в него величин. Методы вывода в классе Console перегружены для всех встроенных типов данных, кроме того, предусмотрены варианты форматного вывода.

[Листинг 4.1](http://www.intuit.ru/department/pl/phlcsharp/3/1.html#example.3.1) содержит два наиболее употребляемых варианта вызова методов вывода. Если метод WriteLine вызван с *одним* параметром, он может быть любого встроенного типа. В строке, помеченной комментариями "1" и "2", нам требуется вывести в каждой строке не одну, а две величины, поэтому прежде чем передавать их для вывода, их требуется "склеить" в одну строку с помощью операции +.

Перед объединением строки с числом надо преобразовать число из его внутренней формы представления в последовательность символов, то есть в строку.*Преобразование в строку* определено во всех стандартных классах C# — для этого служит метод ToString(). В данном случае он выполняется неявно.

Оператор 3 иллюстрирует форматный вывод. В этом случае используется другой вариант метода WriteLine. Первым параметром методу передается строковый литерал, содержащий, помимо обычных символов, предназначенных для вывода на консоль, *параметры* в фигурных скобках, а также *управляющие последовательности*. Параметры нумеруются с нуля, перед выводом они заменяются значениями соответствующих переменных в списке вывода.

Из управляющих последовательностей чаще всего используются символы перевода строки ( \n ) и горизонтальной табуляции ( \t ).

Рассмотрим простейшие способы *ввода с клавиатуры*. В классе Console определены методы ввода строки и отдельного символа, но нет методов, которые позволяют непосредственно считывать с клавиатуры числа. Ввод числовых данных выполняется в два этапа:

1. Символы, представляющие собой число, вводятся с клавиатуры в строковую переменную.
2. Выполняется преобразование из строки в переменную соответствующего типа.

Преобразование можно выполнить либо с помощью специального класса Convert, определенного в пространстве имен System, либо с помощью метода Parse, имеющегося в каждом стандартном арифметическом классе. В [листинге 4.2](http://www.intuit.ru/department/pl/phlcsharp/4/#example.4.2) используются оба способа.

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

Console.WriteLine( "Введите строку" );

string s = Console.ReadLine(); // 1

Console.WriteLine( "s = " + s );

Console.WriteLine( "Введите символ" );

char c = (char)Console.Read(); // 2

Console.ReadLine(); // 3

Console.WriteLine( "c = " + c );

string buf; // строка – буфер для ввода чисел

Console.WriteLine( "Введите целое число" );

buf = Console.ReadLine();

int i = Convert.ToInt32( buf ); // 4

Console.WriteLine( i );

Console.WriteLine( "Введите вещественное число" );

buf = Console.ReadLine();

double x = Convert.ToDouble( buf ); // 5

Console.WriteLine( x );

Console.WriteLine( "Введите вещественное число" );

buf = Console.ReadLine();

double y = double.Parse( buf ); // 6

Console.WriteLine( y );

Console.WriteLine( "Введите вещественное число" );

buf = Console.ReadLine();

decimal z = decimal.Parse( buf ); // 7

Console.WriteLine( z );

}

}

}

**Листинг 4.2. Ввод с консоли (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.2.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.2.txt)**)**

Ввод строки выполняется в операторе 1. Длина строки не ограничена, ввод выполняется до символа перевода строки.

Ввод символа выполняется с помощью метода Read, который считывает один символ из входного потока (оператор 2). Метод возвращает значение типа int, представляющее собой код символа, или –1, если символов во входном потоке нет (например, пользователь нажал клавишу **Enter** ). Поскольку нам требуется неint, а char, а неявного преобразования от int к char не существует, приходится применить операцию явного преобразования типа.

Оператор 3 считывает остаток строки и никуда его не передает. Это необходимо потому, что ввод данных выполняется через *буфер* — специальную область оперативной памяти. Фактически данные сначала заносятся в буфер, а затем считываются оттуда процедурами ввода. Занесение в буфер выполняется по нажатию клавиши **Enter** вместе с ее кодом. Метод Read, в отличие от ReadLine, не очищает буфер, поэтому следующий после него ввод будет выполняться с того места, на котором закончился предыдущий.

В операторах 4 и 5 используются методы класса Convert, в операторах 6 и 7 — методы Parse классов Double и Decimal библиотеки .NET, которые используются здесь через имена типов С# double и decimal.

**Внимание**

*При вводе вещественных чисел дробная часть отделяется от целой с помощью запятой (или точки - зависит от регионанальных настроек)*.

Если вводимые с клавиатуры символы нельзя интерпретировать как вещественное число, генерируется исключение.

### Ввод-вывод в файлы

Часто бывает удобно заранее подготовить исходные данные в текстовом файле и считывать их в программе. Вывод из программы тоже бывает полезно выполнить не на экран, а в текстовый файл. Работа с файлами подробно рассматривается позже, а сейчас приведем лишь образцы для использования в программах. В[листинге 4.3](http://www.intuit.ru/department/pl/phlcsharp/4/#example.4.3) приведена версия программы из [листинга 4.1](http://www.intuit.ru/department/pl/phlcsharp/4/#example.4.1), выполняющая вывод не на экран, а в текстовый файл с именем **output.txt**. Файл создается в том же каталоге, что и исполняемый файл программы, по умолчанию — **...\ConsoleApplication1\bin\Debug**.

using System;

using System.IO; // 1

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

StreamWriter f = new StreamWriter( "output.txt" ); // 2

int i = 3;

double y = 4.12;

decimal d = 600m;

string s = "Вася";

f.WriteLine( "i = " + i ); // 3

f.WriteLine( "s = " + s ); // 4

f.WriteLine( "y = {0} \nd = {1}", y, d ); // 5

f.Close(); // 6

}

}

}

**Листинг 4.3. Вывод в текстовый файл (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.3.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.3.txt)**)**

Для того чтобы использовать в программе файлы, необходимо:

1. Подключить пространство имен, в котором описываются стандартные классы для работы с файлами (оператор 1).
2. Объявить *файловую переменную* и связать ее с файлом на диске (оператор 2).
3. Выполнить операции ввода-вывода (операторы 3–5).
4. Закрыть файл (оператор 6).

Ввод данных из файла выполняется аналогично. В [листинге 4.4](http://www.intuit.ru/department/pl/phlcsharp/4/#example.4.4) приведена программа, аналогичная [листингу 4.2](http://www.intuit.ru/department/pl/phlcsharp/4/#example.4.2), но ввод выполняется из файла с именем **input.txt**, расположенного в каталоге **D:\C#**. Текстовый файл можно создать с помощью любого текстового редактора, удобно использовать Visual Studio.NET.

using System;

using System.IO;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

StreamReader f = new StreamReader( "d:\\C#\\input.txt" );

string s = f.ReadLine();

Console.WriteLine( "s = " + s );

char c = (char)f.Read();

f.ReadLine();

Console.WriteLine( "c = " + c );

string buf;

buf = f.ReadLine();

int i = Convert.ToInt32( buf );

Console.WriteLine( i );

buf = f.ReadLine();

double x = Convert.ToDouble( buf );

Console.WriteLine( x );

buf = f.ReadLine();

double y = double.Parse( buf );

Console.WriteLine( y );

buf = f.ReadLine();

decimal z = decimal.Parse( buf );

Console.WriteLine( z );

f.Close();

}

}

}

**Листинг 4.4. Ввод из текстового файла (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.4.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.4.txt)**)**

### Математические функции — класс Math

Самая простая программа, которую можно себе представить (не считая "Hello, World!"), состоит из ввода исходных данных, вычислений по каким-то формулам и вывода результата. В выражениях, из которых состоят формулы, часто используются математические функции, например, синус или возведение в степень. Они реализованы в классе Math, определенном в пространстве имен System. Описание методов и полей класса приведено в [таблице 4.1](http://www.intuit.ru/department/pl/phlcsharp/4/#table.4.1).

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 4.1. Основные поля и статические методы класса Math | | | |
| Имя | Описание | Результат | Пояснения |
| Abs | Модуль | Перегружен | | *x* | записывается как Abs(x) |
| Acos | Арккосинус | double | Acos(double x) |
| Asin | Арксинус | double | Asin(double x) |
| Atan | Арктангенс | double | Atan2(double x, double y) — угол, тангенс которого есть результат деления y на x |
| BigMul | Произведение | long | BigMul(int x, int y) |
| Ceiling | Округление до большего целого | double | Ceiling(double х) |
| Cos | Косинус | double | Сos(double x) |
| Cosh | Гиперболический косинус | double | Cosh(double x) |
| DivRem | Деление и остаток | Перегружен | DivRem(x, y, rem) |
| E | База натурального логарифма (число е) | double | 2,71828182845905 |
| Exp | Экспонента | double | e *x* записывается как Exp(x) |
| Floor | Округление до меньшего целого | double | Floor(double х) |
| IEEERemainder | Остаток от деления | double | IEEERemainder(double x, double y) |
| Log | Натуральный логарифм | double | loge *x* записывается как Log(x) |
| Log10 | Десятичный логарифм | double | log10 *x* записывается как Log10(x) |
| Max | Максимум из двух чисел | Перегружен | Max(x, y) |
| Min | Минимум из двух чисел | Перегружен | Min(x, y) |
| PI | Значение числа http://www.intuit.ru/img/tex/bddd322d70e234214bfd625a8a7f8f9e.png | double | 3,14159265358979 |
| Pow | Возведение в степень | double | xy записывается как Pow(x, y) |
| Round | Округление | Перегружен | Round(3.1) даст в результате 3 Round (3.8) даст в результате 4 |
| Sign | Знак числа | int | Аргументы перегружены |
| Sin | Синус | double | Sin(double x) |
| Sinh | Гиперболический синус | double | Sinh(double x) |
| Sqrt | Квадратный корень | double | http://www.intuit.ru/img/tex/8758f44c74ca622b741142b34f967534.png *x* записывается как Sqrt(x) |
| Tan | Тангенс | double | Tan(double x) |
| Tanh | Гиперболический тангенс | double | Tanh(double x) |

В [листинге 4.5](http://www.intuit.ru/department/pl/phlcsharp/4/#example.4.5) приведен пример применения методов класса Math.

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

Console.Write( "Введите х: " );

double x = double.Parse( Console.ReadLine() );

Console.Write( "Введите y: " );

double y = double.Parse( Console.ReadLine() );

Console.WriteLine( "Максимум из х и y : " + Math.Max(x, y) );

double z = Math.Pow(Math.Sin(x), 2) + Math.Pow(Math.Sin(y), 2);

Console.WriteLine( "Сумма квадратов синусов х и y : " + z );

}

}

}

**Листинг 4.5. Применение методов класса Math (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.5.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.5.txt)**)**

Теперь вы знаете достаточно, чтобы писать *линейные программы*, все операторы в которых выполняются последовательно, один за другим, например: ввод исходных данных, вычисления по формулам, вывод результатов. Для написания программ более сложной структуры необходимо изучить *управляющие операторы*, которые, как следует из их названия, управляют потоками вычислений.

### Выражения, блоки и пустые операторы

Любое *выражение*, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении выражения. Частным случаем выражения является *пустой оператор* ; (он используется, когда по синтаксису оператор требуется, а по смыслу — нет). Примеры:

i++; // выполняется операция инкремента

a \*= b + c; // выполняется умножение с присваиванием

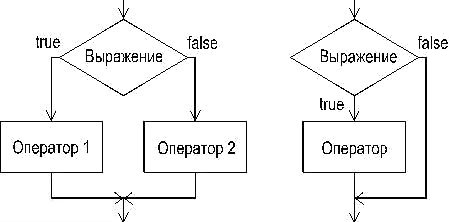
fun( i, k ); // выполняется вызов функции

while( true ); // цикл из пустого оператора (бесконечный)

*Блок*, или *составной оператор*, — это последовательность описаний и операторов, заключенная в фигурные скобки. Блок воспринимается компилятором как один оператор и может использоваться всюду, где синтаксис требует одного оператора, а алгоритм — нескольких. Блок может содержать один оператор или быть пустым.

### Условный оператор if

Условный оператор if используется для разветвления процесса вычислений на два направления. Структурная схема оператора приведена на [рис. 4.1](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#image.4.1).



**Рис. 4.1.** Структурная схема условного оператора

Формат оператора:

if ( логическое\_выражение ) оператор\_1; [ else оператор\_2; ]

Сначала вычисляется логическое выражение. Если оно имеет значение true, выполняется первый оператор, иначе — второй. После этого управление передается на оператор, следующий за условным. Ветвь else может отсутствовать.

Если в какой-либо ветви требуется выполнить несколько операторов, их необходимо заключить в *блок*. Блок может содержать любые операторы, в том числе описания и другие условные операторы, но не может состоять из одних описаний.

Примеры условных операторов:

if ( a < 0 ) b = 1; // 1

if ( a < b && ( a > d || a == 0 ) ) b++; else { b \*= a; a = 0; } // 2

if ( a < b ) if ( a < c ) m = a; else m = c;

else if ( b < c ) m = b; else m = c; // 3

if ( b > a ) max = b; else max = a; // 4

Если требуется проверить несколько условий, их объединяют знаками логических условных операций. Например, выражение *в примере 2* будет истинно в том случае, если выполнится одновременно условие a < b и одно из условий в скобках. Оператор *в примере 3* вычисляет наименьшее значение из трех переменных. Обратите внимание, что компилятор относит часть else к ближайшему ключевому слову if.

В качестве примера подсчитаем количество очков после выстрела по мишени, изображенной на [рис. 4.2](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#image.4.2).

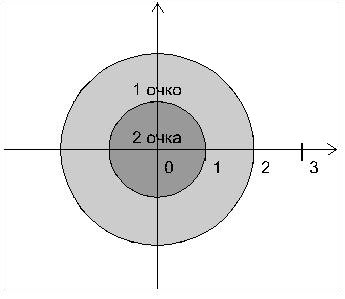


Рис. 4.2. Мишень

Программист выбирает тип переменных, исходя из их назначения. Координаты выстрела нельзя представить целыми величинами, так как это приведет к потере точности результата, а счетчик очков не имеет смысла описывать как вещественный. Программа приведена в [листинге 4.6](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#example.4.6).

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

Console.WriteLine( "Введите координату x" );

double x = Convert.ToDouble(Console.ReadLine() );

Console.WriteLine( "Введите координату у" );

double y = double.Parse( Console.ReadLine() );

int kol = 0;

if ( x \* x + y \* y < 1 ) kol = 2;

else if ( x \* x + y \* y < 4 ) kol = 1;

Console.WriteLine( "Результат = {0} очков", kol );

}

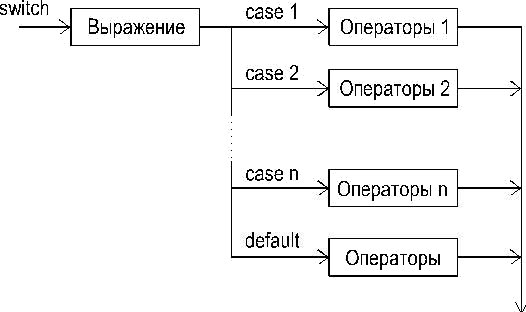
}

}

**Листинг 4.6. Выстрел по мишени (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.6.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.6.txt)**)**

### Оператор выбора switch

Оператор switch (переключатель) предназначен для разветвления процесса вычислений на несколько направлений. Структурная схема оператора приведена на[рис. 4.3](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#image.4.3).



**Рис. 4.3.** Структурная схема оператора switch

Формат оператора:

switch ( выражение ){

case константное\_выражение\_1: [ список\_операторов\_1 ]

case константное\_выражение\_2: [ список\_операторов\_2 ]

...

case константное\_выражение\_n: [ список\_операторов\_n ]

[ default: операторы ]

}

Выполнение оператора начинается с вычисления выражения. Тип выражения чаще всего целочисленный (включая char ) или строковый. Затем управление передается первому оператору из списка, помеченному константным выражением, значение которого совпало с вычисленным. Если совпадения не произошло, выполняются операторы, расположенные после слова default, а при его отсутствии управление передается следующему за switch оператору.

Каждая ветвь переключателя должна заканчиваться явным *оператором перехода*, а именно одним из операторов break, goto или return:

* оператор break выполняет выход из самого внутреннего из объемлющих его операторов switch, for, while и do ;
* оператор goto выполняет переход на указанную после него метку, обычно это метка case одной из нижележащих ветвей оператора switch ;
* оператор return выполняет выход из функции, в теле которой он записан.

В [листинге 4.7](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#example.4.7) приведен пример программы, реализующей простейший калькулятор на четыре действия.

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

string buf;

double a, b, res;

Console.WriteLine( "Введите первый операнд:" );

a = double.Parse( Console.ReadLine() );

Console.WriteLine( "Введите знак операции" );

char op = (char)Console.Read();

Console.ReadLine();

Console.WriteLine( "Введите второй операнд:" );

b = double.Parse( Console.ReadLine() );

bool ok = true;

switch (op)

{

case '+' : res = a + b; break;

case '-' : res = a - b; break;

case '\*' : res = a \* b; break;

case '/' : res = a / b; break;

default : res = double.NaN; ok = false; break;

}

if (ok) Console.WriteLine( "Результат: " + res );

else Console.WriteLine( "Недопустимая операция" );

}

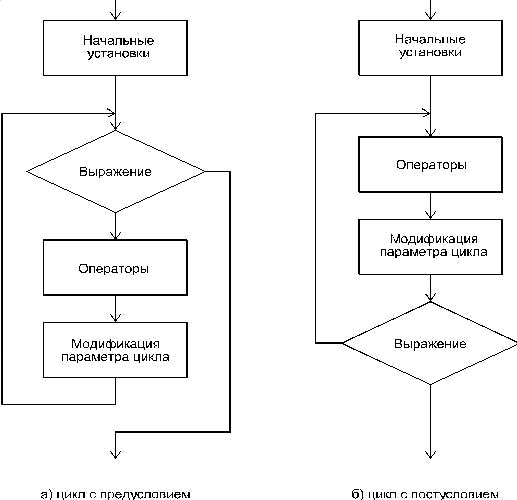
}

}

**Листинг 4.7. Калькулятор на четыре действия (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.7.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.7.txt)**)**

### Операторы цикла и передачи управления

Операторы цикла используются для вычислений, повторяющихся многократно. Блок, ради выполнения которого и организуется цикл, называется *телом цикла*. Остальные операторы служат для управления процессом повторения вычислений: это начальные установки, проверка *условия продолжения цикла* и модификация параметра цикла ([рис. 4.4](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#image.4.4)). Один проход цикла называется *итерацией*.



**Рис. 4.4.** Структурные схемы операторов цикла

*Начальные установки* служат для того, чтобы до входа в цикл задать значения переменных, которые в нем используются.

*Проверка условия* *продолжения цикла* выполняется на каждой итерации либо до тела цикла (тогда говорят о цикле *с предусловием* ), либо после тела цикла (цикл *с постусловием* ).

*Параметром цикла* называется переменная, которая используется при проверке *условия продолжения цикла* и принудительно изменяется на каждой итерации, причем, как правило, на одну и ту же величину. Если параметр цикла целочисленный, он называется *счетчиком цикла*.

Цикл завершается, если условие его продолжения не выполняется. Возможно принудительное завершение как текущей итерации, так и цикла в целом. Для этого служат операторы break, continue, return и goto. Передавать управление извне внутрь цикла запрещается.

### Цикл с предусловием while

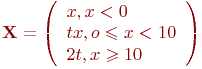
Формат оператора:

while ( выражение ) оператор

Выражение должно быть логического типа. Например, это может быть операция отношения. Если результат вычисления выражения равен true, выполняется простой или составной оператор. Эти действия повторяются до того момента, пока результатом выражения не станет значение false. После окончания цикла управление передается на следующий за ним оператор.

Выражение вычисляется перед каждой итерацией цикла. Если при первой проверке выражение равно false, цикл не выполнится ни разу.

В качестве примера рассмотрим программу, выводящую для аргумента *х*, изменяющегося в заданных пределах с заданным шагом, таблицу значений следующей функции:



Назовем начальное значение аргумента Xn, конечное значение аргумента — Xk, шаг изменения аргумента — dX и параметр t. Все величины вещественные. Программа должна выводить таблицу, состоящую из двух столбцов: значений аргумента и соответствующих им значений функции.

Текст программы приведен в [листинге 4.8](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#example.4.8).

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

double Xn = -2, Xk = 12, dX = 2, t = 2, y;

Console.WriteLine( "| x | y |" ); // заголовок таблицы

double x = Xn;

while ( x <= Xk )

{

y = t;

if ( x >= 0 && x < 10 ) y = t \* x;

if ( x >= 10 ) y = 2 \* t;

Console.WriteLine( "| {0,6} | {1,6} |", x, y );

x += dX;

}

}

}

}

**Листинг 4.8. Таблица значений функции с использованием цикла while (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.8.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.8.txt)**)**

### Цикл с постусловием do

*Цикл с постусловием* реализует структурную схему, приведенную на рис. 4.4, *б*, и имеет вид:

do оператор while ( выражение );

Сначала выполняется простой или составной оператор, образующий тело цикла, а затем вычисляется выражение (оно должно иметь тип bool ). Если выражение истинно, тело цикла выполняется еще раз, и проверка повторяется. Цикл завершается, когда выражение станет равным false или в теле цикла будет выполнен какой-либо *оператор передачи управления*.

Этот вид цикла применяется в тех случаях, когда тело цикла необходимо обязательно выполнить хотя бы один раз. Пример программы, выполняющей проверку ввода, приведен в [листинге 4.9](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#example.4.9).

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

char answer;

do

{

Console.WriteLine( "Купи слоника, а?" );

answer = (char) Console.Read();

Console.ReadLine();

} while ( answer != 'y' );

}

}

}

**Листинг 4.9. Проверка ввода (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.9.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.9.txt)**)**

### Цикл с параметром for

Цикл с параметром имеет следующий формат:

for ( инициализация; выражение; модификации ) оператор;

*Инициализация* служит для объявления величин, используемых в цикле, и присвоения им начальных значений. В этой части можно записать несколько операторов, разделенных запятой, например:

for ( int i = 0, j = 20; ...

int k, m;

for ( k = 1, m = 0; ...

Областью действия переменных, объявленных в части инициализации цикла, является цикл. Инициализация выполняется один раз в начале исполнения цикла.

*Выражение* типа bool определяет условие выполнения цикла: если его результат равен true, цикл выполняется.

*Модификации* выполняются после каждой итерации цикла и служат обычно для изменения параметров цикла. В части модификаций можно записать несколько операторов через запятую, например:

for ( int i = 0, j = 20; i < 5 && j > 10; i++, j-- ) ...

Простой или составной *оператор* представляет собой тело цикла. Любая из частей оператора for может быть опущена (но точки с запятой надо оставить на своих местах!).

Для примера вычислим сумму чисел от 1 до 100:

int s = 0;

for ( int i = 1; i <= 100; i++ ) s += i;

В [листинге 4.10](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#example.4.10) приведена программа, выводящая таблицу значений функции из [листинга 4.8](http://www.intuit.ru/department/pl/phlcsharp/4/2.html#example.4.9).

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

double Xn = -2, Xk = 12, dX = 2, t = 2, y;

Console.WriteLine( "| x | y |"); // заголовок таблицы

for ( double x = Xn; x <= Xk; x += dX )

{

y = t;

if ( x >= 0 && x < 10 ) y = t \* x;

if ( x >= 10 ) y = 2 \* t;

Console.WriteLine( "| {0,6} | {1,6} |", x, y );

}

}

}

}

**Листинг 4.10. Таблица значений функции с использованием цикла for (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.10.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.10.txt)**)**

Любой цикл while может быть приведен к эквивалентному ему циклу for и наоборот. Например, два следующих цикла эквивалентны:

Цикл for:

for ( b1; b2; b3 ) оператор;

Цикл while:

b1;

while ( b2 )

{

оператор;

b3

}

### Цикл перебора foreach

Цикл foreach используется для просмотра всех объектов из некоторой группы данных, например, массива, списка или другого контейнера. Он будет рассмотрен в разделе "Оператор foreach".

### Операторы передачи управления

В С# есть пять операторов, изменяющих естественный порядок выполнения вычислений:

* оператор безусловного перехода goto ;
* оператор выхода из цикла break ;
* оператор перехода к следующей итерации цикла continue ;
* оператор возврата из функции return ;
* оператор генерации исключения throw.

Эти операторы могут передать управление в пределах блока, в котором они использованы, и за его пределы. Передавать управление внутрь другого блока запрещается.

### Оператор goto

Оператор безусловного перехода goto используется в одной из трех форм:

goto метка;

goto case константное\_выражение;

goto default;

В теле той же функции должна присутствовать ровно одна конструкция вида:

метка: оператор;

Оператор goto метка передает управление на помеченный оператор. *Метка* — это обычный идентификатор, областью видимости которого является функция, в теле которой он задан. Метка должна находиться в той же области видимости, что и оператор перехода.

Вторая и третья формы оператора goto используются в *теле оператора* выбора switch. Оператор goto case константное\_выражение передает управление на соответствующую константному выражению ветвь, а оператор goto default — на ветвь default.

### Оператор break

Оператор break используется внутри операторов цикла или выбора для перехода в точку программы, находящуюся непосредственно за оператором, внутри которого находится оператор break.

Для примера рассмотрим программу вычисления значения функции Sin x (синус) с точностью http://www.intuit.ru/img/tex/f92f9a9342b9900692ae6f5b845210b6.png = 10-6 с помощью бесконечного ряда Тейлора по формуле:

y = x - x3/3! + x5/5! - x7/7! +…

Этот ряд сходится при | x | < http://www.intuit.ru/img/tex/311dfd4ffb207575ba41134442fff513.png. Точность достигается при |Rn| < ε, где Rn —остаточный член ряда, который для данного ряда можно заменить величиной Cnочередного члена ряда, прибавляемого к сумме.

Алгоритм решения задачи выглядит так: задать начальное значение *суммы ряда*, а затем многократно вычислять очередной член ряда и добавлять его к ранее найденной сумме. Вычисления заканчиваются, когда абсолютная величина очередного члена ряда станет меньше заданной точности.

Для уменьшения количества выполняемых действий следует воспользоваться рекуррентной формулой получения последующего члена ряда через предыдущий:Cn+1 = Cn x T, где T — некоторый множитель. Подставив в эту формулу Cn и Cn+1, получим выражение для вычисления Т:

http://www.intuit.ru/img/tex/210e300a6fd0e07c16767bb460e67452.png

В [листинге 4.11](http://www.intuit.ru/department/pl/phlcsharp/4/3.html#example.4.11) приведен текст программы с комментариями.

using System;

namespace ConsoleApplication1

{ class Class1

{ static void Main()

{

double e = 10-6;

const int MaxIter = 500; // ограничитель количества итераций

Console.WriteLine( "Введите аргумент:" );

double x = Convert.ToDouble( Console.ReadLine() );

bool done = true; // признак достижения точности

double ch = x, y = ch;

for ( int n = 0; Math.Abs(ch) > e; n++ )

{

ch \*= -x \* x / (2 \* n + 2 ) / ( 2 \* n + 3); // очередной член ряда

y += ch; // добавление члена ряда к сумме

if ( n > MaxIter ) { done = false; break; }

}

if ( done ) Console.WriteLine( "Сумма ряда - " + y );

else Console.WriteLine( "Ряд расходится" );

}

}

}

**Листинг 4.11. Вычисление суммы бесконечного ряда (**[**html**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.11.htm)**,** [**txt**](http://www.intuit.ru/department/pl/phlcsharp/4/example.4.11.txt)**)**

Получение суммы бесконечного ряда — пример вычислений, которые принципиально невозможно выполнить точно.

### Оператор continue

Оператор перехода к следующей итерации текущего цикла continue пропускает все операторы, оставшиеся до конца тела цикла, и передает управление на начало следующей итерации.

Перепишем основной цикл [листинга 4.11](http://www.intuit.ru/department/pl/phlcsharp/4/3.html#example.4.11) с применением оператора continue:

for ( int n = 0; Math.Abs(ch) > e; n++ )

{

ch \*= x \* x / ( 2 \* n + 1 ) / ( 2 \* n + 2 );

y += ch;

if ( n <= MaxIter ) continue;

done = false; break;

### Оператор return

Оператор возврата из функции return завершает выполнение функции и передает управление в точку ее вызова. Синтаксис оператора:

return [ выражение ];

Тип выражения должен иметь неявное преобразование к типу функции. Если тип возвращаемого функцией значения описан как void, выражение должно отсутствовать.

### Базовые конструкции структурного программирования

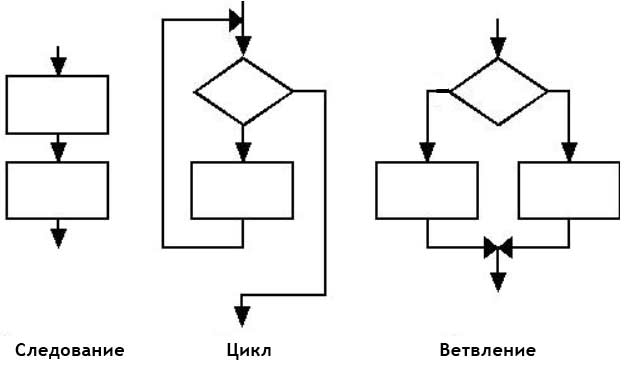
Главное требование, которому должна удовлетворять программа, — работать в полном соответствии со спецификацией и адекватно реагировать на любые действия пользователя. Кроме этого, программа должна быть выпущена точно к заявленному сроку и допускать оперативное внесение необходимых изменений и дополнений.

Иными словами, современные критерии качества программы — это, прежде всего, *надежность*, а также возможность точно *планировать* производство программы и ее *сопровождение*. Для достижения этих целей программа должна иметь простую структуру, быть читабельной и легко модифицируемой. Технология структурного программирования позволяет создавать как раз такие программы небольшого и среднего объема. Для разработки более сложных комплексов требуется применять объектно-ориентированное программирование.

*Структурное программирование* — это технология создания программ, позволяющая путем соблюдения определенных правил сократить время разработки и уменьшить количество ошибок, а также облегчить возможность модификации программы.

В C# идеи структурного программирования используются на самом низком уровне — при написании методов объектов. Доказано, что любой алгоритм можно реализовать только из трех структур, называемых *базовыми конструкциями* структурного программирования: это следование, ветвление и цикл.

*Следованием* называется конструкция, реализующая последовательное выполнение двух или более операторов (простых или составных). *Ветвление* задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия. *Цикл* реализует многократное выполнение оператора. Базовые конструкции приведены на [рис. 4.5](http://www.intuit.ru/department/pl/phlcsharp/4/3.html#image.4.5).



**Рис. 4.5.** Базовые конструкции структурного программирования

Особенностью базовых конструкций является то, что любая из них имеет только один вход и один выход, поэтому конструкции могут вкладываться друг в друга произвольным образом, например, цикл может содержать следование из двух ветвлений, каждое из которых включает вложенные циклы.

*Целью* использования базовых конструкций является получение программы простой структуры. Такую программу легко читать, отлаживать и при необходимости вносить в нее изменения. В большинстве языков высокого уровня существует несколько реализаций базовых конструкций; в С# им соответствует четыре вида циклов и два вида ветвлений.

## Обработка исключительных ситуаций

В языке C# есть операторы, позволяющие обнаруживать и обрабатывать ошибки (исключительные ситуации), возникающие в процессе выполнения программы.

*Исключительная ситуация*, или *исключение*, — это возникновение аварийного события, которое может порождаться некорректным использованием аппаратуры или неправильной работой программы, например, делением на ноль или переполнением. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. С# дает программисту возможность восстановить работоспособность программы и продолжить ее выполнение.

Исключения С# не поддерживают обработку асинхронных событий, таких как ошибки оборудования или прерывания, например, нажатие клавиш **Ctrl** + **C**. Механизм исключений предназначен только для событий, которые могут произойти в результате работы самой программы и указываются явным образом. Исключения возникают тогда, когда некоторая часть программы не смогла сделать то, что от нее требовалось. При этом другая часть программы может попытаться сделать что-нибудь иное.

*Исключения позволяют логически разделить вычислительный процесс на две части* — *обнаружение аварийной ситуации и ее обработка*. Это важно не только для лучшей структуризации программы. Главное то, что функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту *функцию код* может знать, что делать, но не уметь определить место возникновения. Это особенно актуально при использовании библиотечных функций и программ, состоящих из многих модулей.

Исключения генерирует либо среда выполнения, либо программист с помощью оператора throw. В [таблице 4.2](http://www.intuit.ru/department/pl/phlcsharp/4/4.html#table.4.2) приведены наиболее часто используемые стандартные исключения, генерируемые средой. Они определены в пространстве имен System. Все они являются потомками класса Exception, а точнее, потомками его потомка SystemException.

|  |  |
| --- | --- |
| Таблица 4.2. Часто используемые стандартные исключения | |
| Имя | Описание |
| DivideByZeroException | Попытка деления на ноль |
| FormatException | Попытка передать в метод аргумент неверного формата |
| IndexOutOfRangeException | Индекс массива выходит за границы диапазона |
| InvalidCastException | Ошибка преобразования типа |
| OutOfMemoryException | Недостаточно памяти для создания нового объекта |
| OverFlowException | Переполнение при выполнении арифметических операций |
| StackOverFlowException | Переполнение стека |

Исключения обнаруживаются и обрабатываются в операторе try.

### Оператор try

Оператор try содержит три части:

* *контролируемый блок* — составной оператор, предваряемый ключевым словом try. В контролируемый блок включаются потенциально опасные операторы программы. Все функции, прямо или косвенно вызываемые из блока, также считаются ему принадлежащими;
* один или несколько *обработчиков исключений* — блоков catch, в которых описывается, как обрабатываются ошибки различных типов;
* *блок завершения* finally выполняется независимо от того, возникла ошибка в контролируемом блоке или нет.

Синтаксис оператора try:

try блок [ блоки catch ] [ блок finally ]

Отсутствовать могут либо блоки catch, либо блок finally, но не оба одновременно.

Рассмотрим, каким образом реализуется обработка исключительных ситуаций.

1. Обработка исключения начинается с появления ошибки. Функция или операция, в которой возникла ошибка, генерирует исключение.
2. Выполнение текущего блока прекращается, отыскивается соответствующий обработчик исключения, и ему передается управление.
3. Выполняется блок finally, если он присутствует.
4. Если обработчик не найден, вызывается стандартный обработчик исключения. Обычно он выводит на экран окно с информацией об исключении и завершает текущий процесс.

*Обработчики исключений* должны располагаться непосредственно за блоком try. Они начинаются с ключевого слова catch, за которым в скобках следует тип обрабатываемого исключения. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений. Блоки catchпросматриваются в том порядке, в котором они записаны, пока не будет найден соответствующий типу выброшенного исключения.

Существуют три формы записи обработчиков:

catch( тип имя ) { ... /\* тело обработчика \*/ }

catch( тип ) { ... /\* тело обработчика \*/ }

catch { ... /\* тело обработчика \*/ }

*Первая форма* применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий.

*Вторая форма* не предполагает использования информации об исключении, играет роль только его тип.

*Третья форма* применяется для перехвата всех исключений. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа (он может быть только один) следует помещать после всех остальных.

Пример:

try {

... // Контролируемый блок

}

catch ( OverflowException e ) {

... // Обработка исключений класса OverflowException (переполнение)

}

catch ( DivideByZeroException ) {

... // Обработка исключений класса DivideByZeroException (деление на 0)

}

catch {

... // Обработка всех остальных исключений

}

Если исключение в контролируемом блоке не возникло, все обработчики пропускаются.

В любом случае, произошло исключение или нет, управление передается в блок завершения finally (если он существует), а затем — первому оператору, находящемуся непосредственно за оператором try.

Операторы try могут многократно вкладываться друг в друга. Исключение, которое возникло во внутреннем блоке try и не было перехвачено соответствующим блоком catch, передается на верхний уровень, где продолжается поиск подходящего обработчика. Этот процесс называется *распространением исключения*.

### Оператор throw

До сих пор мы рассматривали исключения, которые генерирует среда выполнения C#, но это может сделать и сам программист. Для генерации исключения используется оператор throw с параметром, определяющим вид исключения. Параметр должен быть объектом, порожденным от стандартного классаSystem.Exception. Этот объект используется для передачи информации об исключении его обработчику.

Синтаксис оператора throw:

throw [ выражение ];

Форма без параметра применяется только внутри блока catch для повторной генерации исключения. Тип выражения, стоящего после throw, определяет тип исключения, например:

throw new DivideByZeroException();

Здесь после слова throw записано выражение, создающее объект стандартного класса "ошибка при делении на 0" с помощью операции new.

При генерации исключения выполнение текущего блока прекращается и происходит поиск соответствующего обработчика с передачей ему управления. Обработчик считается найденным, если тип объекта, указанного после throw, либо тот же, что задан в параметре catch, либо является производным от него.