

Состав языка

Алфавит – совокупность допустимых в языке символов. *Алфавит* языка C# включает:

1. прописные и строчные латинские буквы и буквы национальных алфавитов (включая кириллицу);
2. арабские цифры от 0 до 9, шестнадцатеричные цифры от A до F ;
3. специальные знаки: " { } , | ; [] () + - / % * . \ ' : ? < = > ! & ~ ^ @ _
4. пробельные символы: пробел, символ табуляции, символ перехода на новую строку.

Из символов алфавита формируются лексемы языка: идентификаторы, ключевые (зарезервированные) слова, знаки операций, *константы*, разделители (скобки, точка, запятая, пробельные символы).

Идентификатор – это имя программного элемента: *константы*, переменной, метки, типа, класса, объекта, метода и т.д. *Идентификатор* может включать латинские буквы и буквы национальных алфавитов, цифры и символ подчеркивания. Прописные и строчные буквы различаются, например, *myname*, *myName* и *MyName* — три различных имени. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра.

Пробелы внутри имен не допускаются. Язык C# не налагает никаких ограничений на длину имен, однако для удобства чтения и записи кода не стоит делать их слишком длинными.

В нотации *Pascal* каждое *слово*, входящее в *идентификатор*, начинается с заглавной буквы. Например: *Age*, *LastName*, *TimeOfDeath*.

Ключевые слова – это зарезервированные идентификаторы, которые имеют специальное *значение* для компилятора, например, *include*, *main*, *int* и т.д. Ключевые слова можно использовать только по прямому назначению. С ключевыми словами и их назначением можно ознакомиться в справочной системе C#.

Типы данных

В C# типы делятся на две группы: *базовые* типы, предлагаемые языком, и типы, *определяемые пользователем*. Кроме того, типы C# разбиваются на две другие категории: *размерные типы* (типы по значению) и *ссылочные типы*. Почти все базовые типы являются размерными типами. Исключение составляют типы Object и String. Все пользовательские типы, кроме структур, являются ссылочными.

Язык C# предлагает обычный набор базовых типов, каждому из них соответствует тип, поддерживаемый общезыковой спецификацией .NET(CLS).

Тип	Размер в байтах	Тип .NET	Описание
Базовый тип			
object		Object	Может хранить все что угодно, т.к. является всеобщим предком
Логический тип			
bool	1	Boolean	true или false
Целые типы			
sbyte	1	SByte	Целое со знаком (от -128 до 127)
byte	1	Byte	Целое без знака (от 0 до 255)
short	2	Int16	Целое со знаком (от -32768 до 32767)
ushort	2	UInt16	Целое без знака (от 0 до 65535)
int	4	Int32	Целое со знаком (от -2147483648 до 2147483647)
uint	4	UInt	Целое число без знака (от 0 до 4 294 967 295)
long	8	Int64	Целое со знаком (от -9223372036854775808 до 9223372036854775807)
ulong	8	UInt64	Целое без знака (от 0 до 0xffffffffffffff)

Вещественные типы			
float	4	Single	Число с плавающей точкой двойной точности. Содержит значения приблизительно от $-1.5 \cdot 10^{-45}$ до $+3.4 \cdot 10^{38}$ с 7 значащими цифрами
double	8	Double	Число с плавающей точкой двойной точности. Содержит значения приблизительно от $-5.0 \cdot 10^{-324}$ до $-1.7 \cdot 10^{308}$ с 15-16 значащими цифрами
Символьный тип			
char	2	Char	Символы Unicode
Строковый тип			
string		String	Строка из Unicode-символов
Финансовый тип			
decimal	12	Decimal	Число до 28 знаков с фиксированным положением десятичной точки. Обычно используется в финансовых расчетах. Требуется суффикс <<m>> или <<M>>

Переменные и константы

Переменная представляет собой типизированную область памяти. Программист создает переменную, объявляя ее тип и указывая имя. При объявлении переменной ее можно инициализировать (присвоить ей начальное *значение*), а затем в любой момент ей можно присвоить новое *значение*, которое заменит собой предыдущее.

```
static void Main()
{
    int i=10;    //объявление и инициализация целочисленной переменной i
    Console.WriteLine(i);    //просмотр значения переменной
    i=100;    //изменение значение переменной
    Console.WriteLine(i);
}
```

В языке C# требуется, чтобы переменные были явно проинициализированы до их использования. Проверим этот факт на примере.

```
static void Main()
{
    int i;
    Console.WriteLine(i);
}
```

При попытке скомпилировать этот пример в списке ошибок будет выведено следующее сообщение: *Use of unassigned local variable 'i'* (используется неинициализированная локальная

Перечисления (enumerations) являются альтернативой константам. *Перечисление* - это особый размерный тип, состоящий из набора именованных констант (называемых *списком перечисления*).

Синтаксис определения перечисления следующий:

```
[атрибуты] [модификаторы] enum <имя> [ : базовый тип]
{список-перечисления констант(через запятую)};
```

Базовый тип - это тип самого перечисления. Если не указать *базовый тип*, то по умолчанию будет использован тип *int*. В качестве базового типа можно выбрать любой *целый* тип, кроме *char*. Пример использования перечисления:

```
class Program
{
    enum gradus:int
    {
        min=0,
        krit=72,
        max=100,
    }
    static void Main()
    {
        Console.WriteLine("минимальная температура=" + (int) gradus.min);
        Console.WriteLine("критическая температура=" + (int)gradus.krit);
        Console.WriteLine("максимальная температура=" + (int)gradus.max);
    }
}
```

Замечания

1. Запись (int) gradus.min используется для явного преобразования перечисления к целому типу. Если убрать (int), то на экран будет выводиться название констант.

2. Символ + в записи "минимальная температура=" + (int) gradus.min при обращении к методу WriteLine означает, что строка "минимальная температура=" будет "склеена" со строковым представлением значения (int) gradus.min. В результате получится новая строка, которая и будет выведена на экран.

Организация ввода-вывода данных. Форматирование

Для обмена данными с внешними устройствами используются специальные объекты. В частности, для работы с консолью используется стандартный *класс* Console, определенный в пространстве имен System.

Вывод данных

В приведенных выше примерах мы уже рассматривали метод WriteLine, реализованный в классе Console, который позволяет организовывать вывод данных на экран. Существует несколько способов применения данного метода:

1. Console.WriteLine(x); //на экран выводится значение идентификатора x
2. Console.WriteLine("x=" + x + "y=" + y); /* на экран выводится строка, образованная последовательным слиянием строки "x=", значения x, строки "y=" и значения y */
3. Console.WriteLine("x={0} y={1}", x, y); /* на экран выводится строка, формат которой задан первым аргументом метода, при этом вместо параметра {0} выводится значение x, а вместо {1} – значение y */

Рассмотрим следующий фрагмент программы:

```
int i=3, j=4;  
Console.WriteLine("{0} {1}", i, j);
```

При обращении к методу WriteLine через запятую перечисляются три аргумента: "{0} {1}", i, j. Первый аргумент определяет формат выходной строки. Следующие аргументы нумеруются с нуля, так переменная i имеет номер 0, j – номер 1. Значение переменной i будет помещено в выходную строку на место {0}, а значение переменной j – на место {1}. В результате на экран будет выведена строка: 3 4.

Если мы обратимся к методу WriteLine следующим образом Console.WriteLine("{0} {1} {2}", j, i, j), то на экран будет выведена строка: 4 3 4.

Последний вариант использования метода WriteLine является наиболее универсальным, потому что он позволяет не только выводить данные на экран, но и управлять форматом их вывода. Рассмотрим несколько примеров:

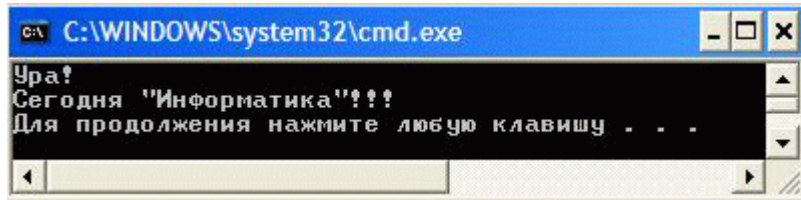
1. *Использование управляющих последовательностей:*

Управляющей последовательностью называют определенный символ, предваряемый обратной косой чертой. Данная совокупность символов интерпретируется как одиночный символ и используется для представления кодов символов, не имеющих графического обозначения (например, символа перевода курсора на новую строку) или символов, имеющих специальное обозначение в символьных и строковых константах (например, апостроф). Рассмотрим управляющие символы:

Вид	Наименование
\a	Звуковой сигнал
\b	Возврат на шаг назад
\f	Перевод страницы
\n	Перевод строки
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\\	Обратная косая черта
\'	Апостроф
\"	Кавычки

Пример:

```
static void Main()  
{  
    Console.WriteLine("Ура!\nСегодня \"Информатика\"!!!");  
}
```

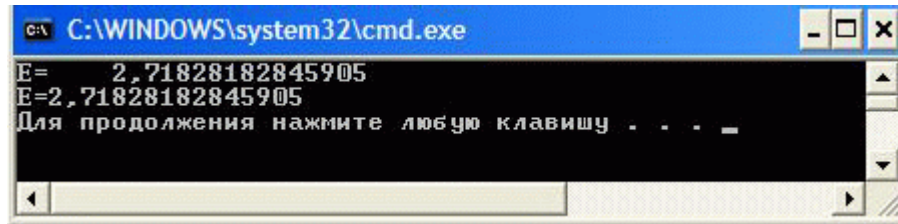


2. *Управление размером поля вывода:*

Первым аргументом `WriteLine` указывается строка вида `{n, m}` – где `n` определяет номер идентификатора из списка аргументов метода `WriteLine`, а `m` – количество позиций (размер поля вывода), отводимых под значение данного идентификатора. При этом значение идентификатора выравнивается по правому краю. Если выделенных позиций для размещения значения идентификатора окажется недостаточно, то автоматически добавится необходимое количество позиций.

Пример:

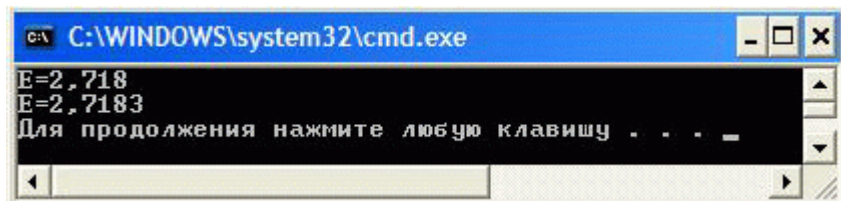
```
static void Main()
{
    double x= Math.E;
    Console.WriteLine("E={0,20}", x);
    Console.WriteLine("E={0,10}", x);
}
```



3. Управление размещением вещественных данных:

Первым аргументом `WriteLine` указывается строка вида `{n: ##.###}` – где `n` определяет номер идентификатора из списка аргументов метода `WriteLine`, а `##.###` определяет *формат вывода* вещественного числа. В данном случае под целую часть числа отводится две позиции, под дробную – три. Если выделенных позиций для размещения целой части значения идентификатора окажется недостаточно, то автоматически добавиться необходимое количество позиций. Пример:

```
static void Main()
{
    double x= Math.E;
    Console.WriteLine("E={0:##.###}", x);
    Console.WriteLine("E={0:#####}", x);
}
```



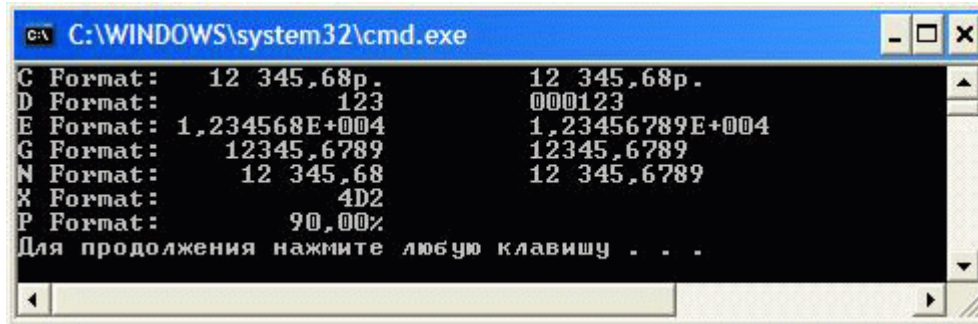
4. Управление форматом числовых данных:

Первым аргументом WriteLine указывается строка вида {n: <спецификатор>m} – где n определяет номер идентификатора из списка аргументов метода WriteLine, <спецификатор> - определяет формат данных, а m – количество позиций для дробной части значения идентификатора. В качестве спецификаторов могут использоваться следующие значения:

Параметр	Формат	Значение
C или c	Денежный. По умолчанию ставит знак р. Изменить его можно с помощью объекта NumberFormatInfo	Задается количество десятичных разрядов.
D или d	Целочисленный (используется только с целыми числами)	Задается минимальное количество цифр. При необходимости результат дополняется начальными нулями
E или e	Экспоненциальное представление чисел	Задается количество символов после запятой. По умолчанию используется 6
F или f	Представление чисел с фиксированной точкой	Задается количество символов после запятой
G или g	Общий формат (или экспоненциальный, или с фиксированной точкой)	Задается количество символов после запятой. По умолчанию выводится целая часть
N или n	Стандартное форматирование использованием запятых и пробелов в качестве разделителей между разрядами	Задается количество символов после запятой. По умолчанию – 2, если число целое, то ставятся нули
X или x	Шестнадцатеричный формат	
P или p	Процентный	

Пример:

```
static void Main()
{
    Console.WriteLine("C Format:{0,14:C} \t{0:C2}", 12345.678);
    Console.WriteLine("D Format:{0,14:D} \t{0:D6}", 123);
    Console.WriteLine("E Format:{0,14:E} \t{0:E8}", 12345.6789);
    Console.WriteLine("G Format:{0,14:G} \t{0:G10}", 12345.6789);
    Console.WriteLine("N Format:{0,14:N} \t{0:N4}", 12345.6789);
    Console.WriteLine("X Format:{0,14:X} ", 1234);
    Console.WriteLine("P Format:{0,14:P} ", 0.9);
}
```



```
C:\WINDOWS\system32\cmd.exe
C Format: 12 345,68p.      12 345,68p.
D Format: 123             000123
E Format: 1,234568E+004    1,23456789E+004
G Format: 12345,6789      12345,6789
N Format: 12 345,68       12 345,6789
X Format: 4D2
P Format: 90,00%
Для продолжения нажмите любую клавишу . . .
```

Ввод данных

Для ввода данных обычно используется метод `ReadLine`, реализованный в классе `Console`. Особенностью данного метода является то, что в качестве результата он возвращает строку (`string`).

Пример:

```
static void Main()
{
    string s = Console.ReadLine();
    Console.WriteLine(s);
}
```

Для того чтобы получить числовое значение необходимо воспользоваться преобразованием данных. Пример:

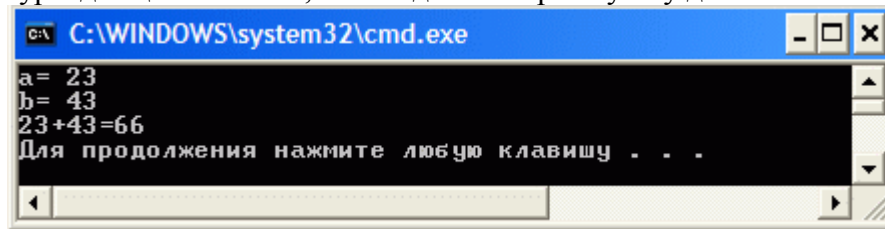
```
static void Main()
{
    string s = Console.ReadLine();
    int x = int.Parse(s); //преобразование строки в число
    Console.WriteLine(x);
}
```

Или сокращенный вариант:

```
static void Main()
{
    //преобразование введенной строки в число
    int x = int.Parse(Console.ReadLine());
    Console.WriteLine(x);
}
```

Для преобразования строкового представления целого числа в тип `int` мы используем метод `int.Parse()`, который реализован для всех числовых типов данных. Таким образом, если нам потребуется преобразовать строковое представление в вещественное, мы можем воспользоваться методом `float.Parse()` или `double.Parse()`. В случае, если соответствующее преобразование выполнить невозможно, то выполнение программы прерывается и генерируется исключение `System.FormatException` (*входная строка* имела неверный формат).

Пример. Написать программу, которая, реализует диалог с пользователем, запрашивает с клавиатуры два целых числа, и выводит на экран сумму данных чисел:



```
using System;
namespace Hello
{
    class Program
    {
        static void Main()
        {
            Console.Write("a= ");
            int a = int.Parse(Console.ReadLine());
            Console.Write("b= ");
            int b = int.Parse(Console.ReadLine());
            Console.WriteLine("{0}+{1}={2}", a, b, a + b);
        }
    }
}
```

Операции

1. Инкремент (++) и декремент(--).

Эти операции имеют две формы записи - *префиксную*, когда операция записывается перед операндом, и *постфиксную* - операция записывается после операнда. *Префиксная операция* инкремента (декремента) увеличивает (уменьшает) свой операнд и возвращает измененное значение как результат. Постфиксные версии инкремента и декремента возвращают первоначальное значение операнда, а затем изменяют его.

<pre>static void Main() { int i = 3, j = 4; Console.WriteLine("{0} {1}", i, j); Console.WriteLine("{0} {1}", ++i, --j); Console.WriteLine("{0} {1}", i++, j--); Console.WriteLine("{0} {1}", i, j); }</pre>	<p><i>Результат работы программы:</i></p> <pre>3 4 4 3 4 3 5 2</pre>
---	--

Замечание. Префиксная версия требует существенно меньше действий: она изменяет значение переменной и запоминает результат в ту же переменную. Постфиксная операция должна отдельно сохранить исходное значение, чтобы затем вернуть его как результат. Для сложных типов подобные дополнительные действия могут оказаться трудоемки. Поэтому постфиксную форму имеет смысл

использовать только при необходимости.

2. **Операция new.** Используется для создания нового объекта. С помощью ее можно создавать как объекты ссылочного типа, так и размерные, например:

3. `object z=new object();`
`int i=new int();` // то же самое, что и `int i =0;`

4. **Отрицание:**

- Арифметическое отрицание (-) - меняет знак операнда на противоположный.
- Логическое отрицание (!) - определяет операцию инверсия для логического типа.

<pre>static void Main() { int i = 3, j=-4; bool a = true, b=false; Console.WriteLine("{0} {1}", -i, -j); Console.WriteLine("{0} {1}", !a, !b); }</pre>	<p><i>Результат работы программы:</i></p> <p>-3 4 False True</p>
--	--

5. **Явное преобразование типа.** Используется для явного преобразования из одного типа в другой. Формат операции:

(тип) выражение;

<pre>static void Main() { int i = -4; byte j = 4; int a = (int)j; //преобразование без потери точности byte b = (byte)i; //преобразование с потерей точности Console.WriteLine("{0} {1}", a, b); }</pre>	<p><i>Результат работы программы:</i></p> <p>4 252</p>
--	--

6. **Умножение (*), деление (/) и деление с остатком (%).** Операции умножения и деления применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно *неявное преобразование* к целым или вещественным типам. При этом тип результата равен "наибольшему" из типов операндов, но не менее int. Если оба операнда при делении целочисленные, то и результат тоже целочисленный.

```
static void Main()
{
    int i = 100, j = 15;
    double a = 14.2, b = 3.5;
    Console.WriteLine("{0} {1} {2}", i*j, i/j, i%j);
    Console.WriteLine("{0} {1} {2}", a * b, a / b, a % b);
}
```

Результат работы программы:

```
1500  6  10
49.7  4.05714285714286  0.19999999999999999
```

7. **Сложение (+) и вычитание (-).** Операции сложения и вычитания применимы для целочисленных и вещественных типов данных. Для других типов эти операции применимы, если для них возможно *неявное преобразование* к целым или вещественным типам.

8. **Операции отношения** (<, <=, >, >=, ==, !=). Операции отношения сравнивают значения левого и правого операндов. Результат операции логического типа: true - если значения совпадают, false - в противном случае.

static void Main() { int i = 15, j = 15; Console.WriteLine(i<j); //меньше Console.WriteLine(i<=j); //меньше или равно Console.WriteLine(i>j); //больше Console.WriteLine(i>=j); //больше или равно Console.WriteLine(i==j); //равно Console.WriteLine(i!=j); //не равно }	<i>Результат работы программы:</i> False True False True True False True False
--	--

1. **Логические операции:** И (&&), ИЛИ (||).

Логические операции применяются к операндам логического типа.

Результат логической операции И имеет значение истина тогда и только тогда, когда оба операнда принимают значение истина.

Результат логической операции ИЛИ имеет значение истина тогда и только тогда, когда хотя бы один из операндов принимает значение истина.

```
static void Main()
{
```

```
Console.WriteLine("x   y   x и y   x или y");  
Console.WriteLine("{0} {1} {2} {3}", false, false, false&&false, false||false);  
Console.WriteLine("{0} {1} {2} {3}", false, true, false&&true, false||true);  
Console.WriteLine("{0} {1} {2} {3}", true, false, true&&false, true||false);  
Console.WriteLine("{0} {1} {2} {3}", true, true, true&&true, true||true);  
}
```

Результат работы программы:

x	y	x и y	x или y
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

2. Условная операция.

Формат: (<операнд1>)? <операнд2> : <операнд3> ;

Операнд1 - это логическое выражение, которое оценивается с точки зрения его эквивалентности константам true и false. Если результат вычисления операнда1 равен true, то результатом условной операции будет значение операнда2, иначе - операнда3. Фактически условная операция является сокращенной формой условного оператора if, который будет рассмотрен позже.

```
static void Main()
{
    int x=5; int y=10;
    int max = (x > y) ? x : y;
    Console.WriteLine(max);
}
```

3. Операции присваивания: =, +=, -= и т.д.

Формат операции *простого присваивания* (=):

операнд_2 = операнд_1;

В результате выполнения этой операции вычисляется значение операнда_1, и результат записывается в операнд_2. Возможно связать воедино сразу несколько операторов присваивания, записывая такие цепочки: a=b=c=100. Выражение такого вида выполняется справа налево: результатом выполнения c=100 является число 100, которое затем присваивается переменной b, результатом чего опять является 100, которое присваивается переменной a.

Кроме простой операции присваивания существуют *сложные операции присваивания*, например, умножение с присваиванием ($*=$), деление с присваиванием ($/=$), остаток от деления с присваиванием ($\%=$), сложение с присваиванием ($+=$), вычитание с присваиванием ($-=$) и т.д.

В сложных операциях присваивания, например, при *сложении с присваиванием*, к операнду_2 прибавляется операнд_1, и результат записывается в операнд_2. То есть, выражение $c += a$ является более компактной записью выражения $c = c + a$. Кроме того, сложные операции присваивания позволяют сгенерировать более эффективный код, за счет того, что в простой операции присваивания для хранения значения правого операнда создается временная переменная, а в сложных операциях присваивания значение правого операнда сразу записывается в левый операнд.

Рассмотренные *операции* приведены с учетом убывания приоритета. Если в одном выражении соседствуют *операции* одного приоритета, то *операции* присваивания и условная *операции* выполняются справа налево, а остальные наоборот. Если необходимо изменить порядок выполнения операций, то в выражении необходимо поставить круглые скобки.

Выражения и преобразование типов

Выражение - это синтаксическая *единица* языка, определяющая способ вычисления некоторого значения. Выражения состоят из операндов, операций и скобок. Каждый *операнд* является в свою очередь выражением или одним из его частных случаев - константой, переменной или функций.

$(a + 0.12)/6$

$x \ \&\& \ y \ || \ !z$

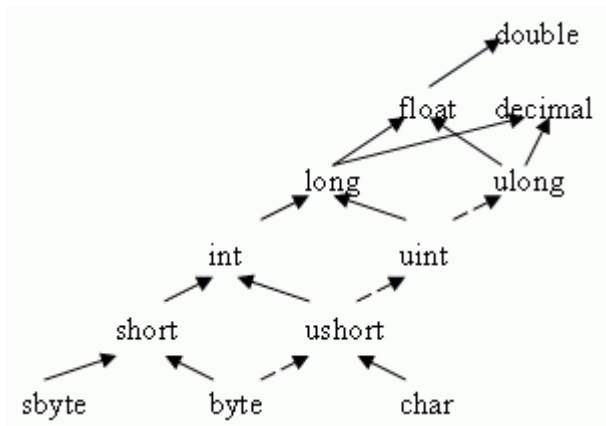
$(t * \text{Math.Sin}(x) - 1.05e4) / ((2 * k + 2) * (2 * k + 3))$

Операции выполняются в соответствии с приоритетами. Для изменения порядка выполнения операций используются круглые скобки. Если в одном выражении записано несколько операций одинакового приоритета, то унарные *операции*, условная операция и *операции* присваивания выполняются *справа налево*, остальные - *слева направо*. Например,

$a = b = c$ означает $a = (b = c)$,

$a + b + c$ означает $(a + b) + c$.

В *выражение* могут входить операнды различных типов. Если операнды имеют одинаковый тип, то результат *операции* будет иметь тот же тип. Если операнды разного типа, то перед вычислениями выполняются преобразования более коротких типов в более длинные для сохранения значимости и точности. *Иерархия типов* данных приведена в следующей схеме:



Преобразование типов в выражениях происходит *неявно* (без участия программистов) следующим образом: 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, то возникает ошибка компиляции (чтобы ее избежать, необходимо воспользоваться операцией явного преобразования). Если путей преобразования несколько, то выбирается наиболее короткий, не содержащий пунктирных линий.

Пример. Написать программу, которая подсчитывает периметр квадрата, площадь которого равна a .

```
using System;
```

```
namespace Example
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main()
```

```
        {
```

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

```
            float s = float.Parse(Console.ReadLine());
```

```
            double p = 4 * Math.Sqrt(s);
```

```
            Console.WriteLine("p=" + p);
```

```
        }
```

```
    }
```

```
}
```

Пример. Написать программу, которая определяет максимальное значение для двух различных вещественных чисел.

```
using System;

namespace Hello
{
    class Program
    {
        static void Main()
        {
            Console.Write("a= "); float a = float.Parse(Console.ReadLine());
            Console.Write("b= "); float b = float.Parse(Console.ReadLine());
            float max=(a>b)?a:b;
            Console.WriteLine("max=" + max);
        }
    }
}
```

Операторы языка C#

Программа на языке C# состоит из последовательности операторов, каждый из которых определяет законченное описание некоторого действия и заканчивается точкой с запятой. Все операторы можно разделить на 4 группы: *операторы* следования, *операторы* ветвления, *операторы* цикла и *операторы* передачи управления.

Операторы следования

Операторы следования выполняются компилятором в естественном порядке: начиная с первого до последнего. К операторам следования относятся: выражение и составной оператор.

Любое *выражение*, завершающееся точкой с запятой, рассматривается как оператор, выполнение которого заключается в вычислении значения выражения или выполнении законченного действия, например, вызова метода.

<code>++i;</code>	<code>//оператор инкремента</code>
<code>x+=y;</code>	<code>//оператор сложение с присваиванием</code>
<code>Console.WriteLine(x);</code>	<code>//вызов метода</code>
<code>x=Math.Pow(a,b)+a*b;</code>	<code>//вычисление сложного выражения</code>

Составной оператор или *блок* представляет собой последовательность операторов, заключенных в фигурные скобки `{}`. Составные операторы применяются в случае, когда правила языка предусматривают наличие только одного оператора, а логика программы требует нескольких операторов. Например, тело цикла `while` должно состоять только из одного оператора. Если заключить несколько операторов в фигурные скобки, то получится блок, который будет рассматриваться компилятором как единый оператор.