Информатика

Язык C#. Типы, операции, структуры управления

© Марченко Антон Александрович 2016 г.
Абрамский Михаил Михайлович

Программа и данные

- Обрабатывает входные данные, генерирует выходные данные
- Должна иметь доступ к данным, которые где-то хранятся
- Должна иметь разные инструменты работы с данными разных типов
  - текст, числа…

Виды памяти

- Внутренняя – просто память
  - для хранения данных во время работы программы
- Внешняя
  - считывается, обрабатывается в программе, записывается
    - примеры: жесткий диск, флешка, DVD диск…
  - продолжительность хранения информации не привязана к работе программы

Хранение данных

- В ячейках памяти
  - с именем для доступа к данным (**имя**)
  - с достаточным местом для хранения (**тип**)
  - значения могут изменяться в процессе работы
    - изменяться по-английски? (не change)

Переменная

- **Имя**
  - идентификатор
    - регулярное выражение [A-Za-z_][A-Za-z0-9_]*
    - начинается с _ или буквы, может содержать цифры
- **Тип**
  - не просто характеризует переменную
    - теория программирования: определяет область значений переменной
    - архитектура: определяет размер данных, хранящихся в переменной

Не совсем «просто память»

- **Физическая** память – оперативная (RAM) память компьютера.
  - размер – несколько ГБ (от 4 до 16 обычно)
  - предназначена для хранения данных
  - ОС и программы считывают и записывают данные в неё пока работает компьютер
- **Виртуальная** память – ведёт себя как физическая, но таковой не является
  - Приложения могут считывать и записывать данные в неё, не беспокоясь о конфликтах с другими приложениями
  - Размер виртуальной памяти может быть больше или меньше физической памяти

Память в .NET Framework

- Стек вызовов (stack)
  - хранит информацию о вызываемых методах (функциях)
  - данные хранятся пока «живёт» метод
- Управляемая куча (managed heap)
  - хранит информацию об объектах
  - данные хранятся пока их не очистит сборщик мусора (Garbage Collector)

Стек - как стопка коробок

- Принцип LIFO (Last In First Out)
- Можно использовать только верхнюю
- Когда верхняя не нужна - выбрасываем её, вершиной становится коробка под ней
- Новая коробка ставится наверх и становится вершиной

Стек вызовов

- Размер **фиксирован** – 1МБ (32bit), 4МБ (64bit)
  - память выделяется один раз при создании потока
- Коробки – вызываемые методы
  - в коробке – переменные
    - аргументы, локальные переменные, возвращаемое значение
- Формальное название коробки – стек фрейм
- Стек - очень **быстр**!
  - добавление/удаление – просто перемещение указателя на вершину стека
- Статический
  - размеры и типы данных определяются при компиляции

Куча – разложенные на полу коробки

- Можно обратиться к любой коробке
- Если коробка больше не нужна – её уберёт уборщик
- У каждой коробки свой адрес в куче

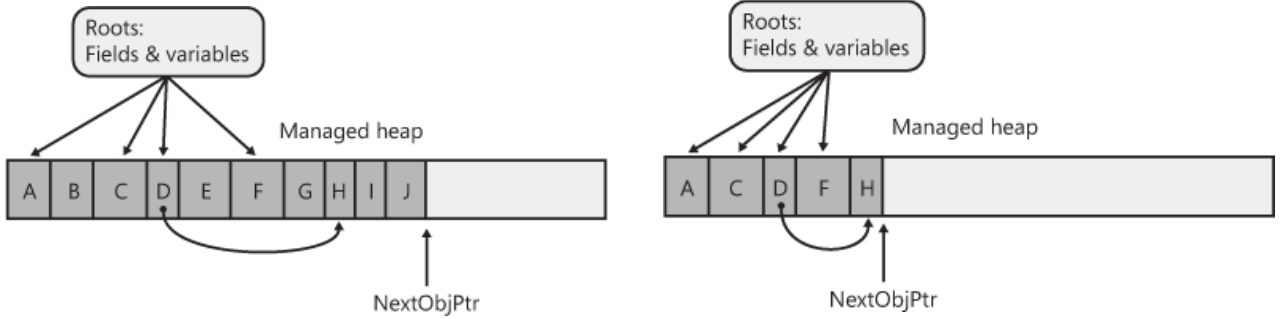
Управляемая куча

- Хранится в виртуальной памяти
- Коробки – отдельные объекты
- **Большого размера** ~1.5ГБ (32bit), ~8ТБ!!! (64bit)
  - запрашивается у ОС по мере необходимости
- Одна на всё приложение
  - а не на поток (проблемы с асинхронным доступом)
- **Медленная**!
  - память выделяется каждый раз при добавлении
  - очищается сборщиком мусора, чья работа требует времени
- Динамическая
  - размер данных определяется при выполнении

Выделение и очищение памяти в куче

- Данные хранятся последовательно друг за другом
- Отслеживается адрес начала свободного места

- При выделении памяти запрашивается виртуальная память и данные добавляются в начало свободного места
- При очищении данные переставляются



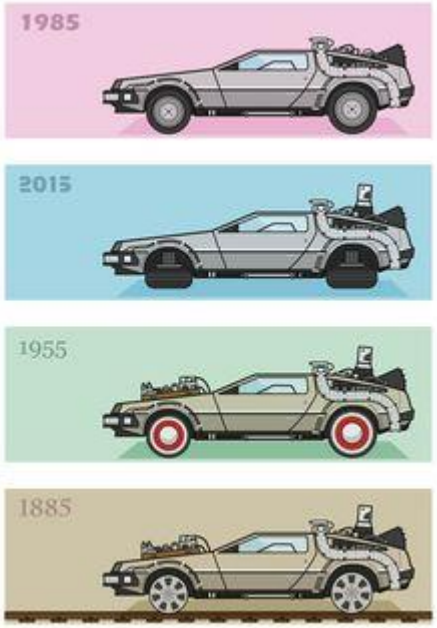
Значения и ссылки

В C# есть два вида типов данных (по способу хранения в памяти):

- ***Типы значения***
  - переменная хранит значение непосредственно в выделенной под неё памяти
  - при копировании – копируются значения
  - значение по умолчанию 0 *('\0', false)*
  - удаляется вместе с содержащим его контейнером
- ***Ссылочные типы***
  - в переменной записана ссылка на значение, само значение - в куче
  - при копировании – копируются ссылки
  - значение по умолчанию *null* (пустая ссылка)
  - удаляется сборщиком мусора, если нет ссылок на значение

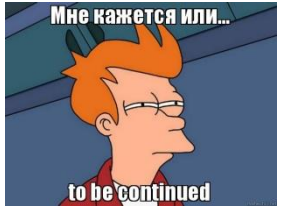
Ссылочные типы

- Указатели
- Интерфейсы
- Массивы
- Строки (string)
- Классы
  - Пользовательские классы
  - Упакованные типы значений
  - Делегаты
- **Обо всём этом позже…**



Типы значения

- **Перечисления**
  - значения – ограниченный список идентификаторов
  - реализуется при помощи целых чисел
  - для контроля над типобезопасностью
  - `enum Cardsuit { Clubs, Diamonds, Spades, Hearts }`
- **Структуры**
  - *примитивные типы* (встроенные)
  - пользовательские типы *(о них позже...)*
    - классы без наследования, конструируются по умолчанию, поля нужно заполнять вручную



Примитивные типы

- *13 типов значений в основе языка*
  - нечисловые
    - *Логический,* *1Байт* bool
    - *Символьный,* *2Байта* char
  - числовые
    - *Целый,* *1Байт* sbyte byte
    - *Целый,* *2Байта* short ushort
    - *Целый,* *4Байта* int uint
    - *Целый,* *8Байт* long ulong
    - *Вещественный,* *4Байта* float
    - *Вещественный,* *8Байт* double
    - *Десятичный,* *16Байт* decimal

Нечисловые типы

- Логический тип ***bool***
  - Используется в логических выражениях
  - Два значения True, False
  - Занимает 1Байт (8 нулей – False, остальное – True)
  - Нет автоматических преобразований в другие типы

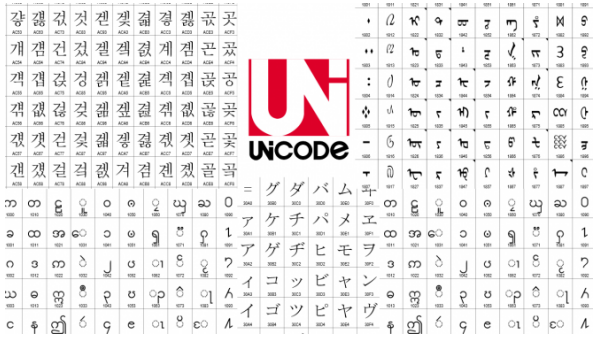
George Boole (1815–1864)

A	B	A • B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

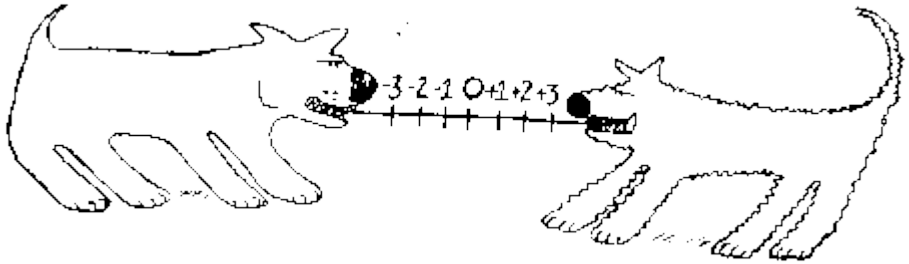
Нечисловые типы

- Символьный тип ***char***
  - Значению соответствует символ
  - Занимает 2Байта
  - 16-разрядный код (Unicode) [0..65 535]
  - ASCII коды – подмножество [0..127]



Числовые типы

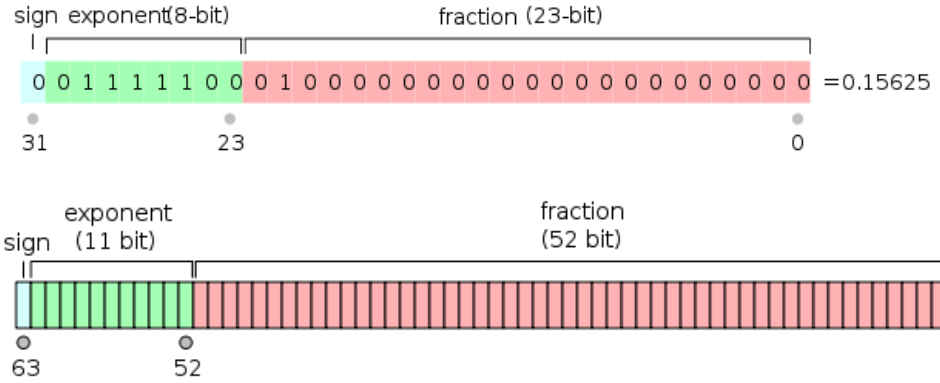
- Целочисленные – со знаком и без от 1Б до 8Б
  - Пример:
    - byte
      - 1Б (8бит)
      - 2^8 значений
      - [0, 255]
- Числа со знаком
  - Пример:
    - sbyte
      - 1Б (8бит, из них 1 – знак)
      - 2^8 значений
      - [-128, 127]



00000000 = 0
00000001 = 1
…
01111111 = 127
10000000 = 128 (вместо 128)
…
11111110 = -2
11111111 = -1 (вместо 255)

Числовые типы

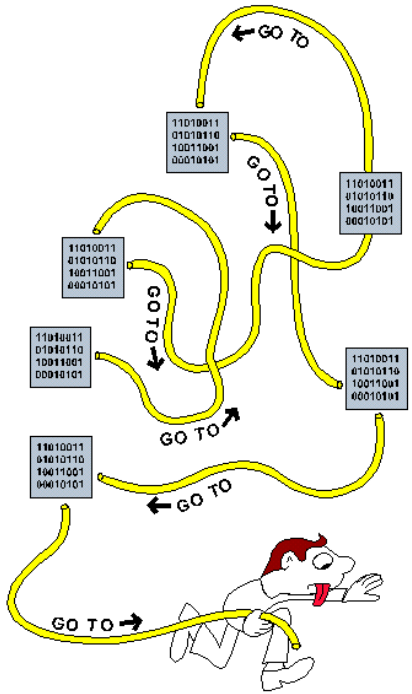
- Вещественные
  - используется экспоненциальная запись числа
  - float
    - 4Байта (32бита):
    - 7-8 цифр
  - double
    - 8Байт (64бита):
    - 15-16 цифр
  - decimal
    - 16Байт(128бит)
    - 28-29 цифр



Поток управления

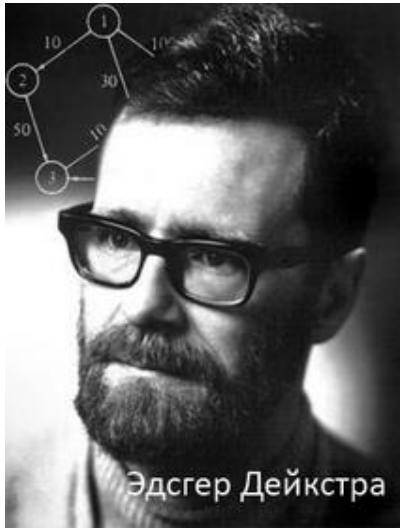
Порядок выполнения команд программы

- Команды
  - *последовательные*:
    не меняют порядок выполнения
  - *переходы*:
    изменяют порядок выполнения
    - безусловные
    - условные



goto considered harmful

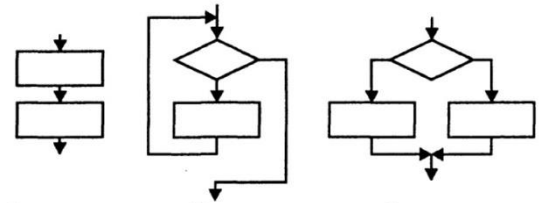
- Много переходов в коде – спагетти код
- *Эдсгер Дейкстра «О вреде оператора goto» (1968)*
  - не использовать goto ЯП высокого уровня!
  - структура программы должна отражалась в порядке выполнения
  - программа – композиция базовых конструкций с одним выходом и одним входом
  - блоки и подпрограммы
  - разработка сверху вниз



Структурное программирование

Э. Дейкстра и др. (1960-е – 1970-е)

- Любая программа может быть постоена из трёх базовых структур управления:
  - **следование**
  - **ветвление**
  - **цикл**
- Теорема Бёма-Якопини (1966)



Базовые конструкции структурного программирования

Операторы в C#

- **Объявления**
  - новой переменной или константы
- **Выражения**
  - вычисление значения
  - присваивание \ вызов метода \ создание объекта
- **Условные**
  - выполнение блоков операторов в зависимости от условия
  - if - else, switch - case
- **Циклические**
  - повторение блоков операторов в зависимости от условия
  - do - while, for, foreach - in
- **Переходы**
  - передача управления в другую часть кода
  - break, continue, default, goto, return, yield
- + Обработка исключений, checked/unchecked, await, yield return, fixed, lock
  но об этом позже...

Операторы

Приоритет	Категория	Операторы	Порядок
0	**Основные**	x.Y f(x) a[x] x++ x-- new checked unchecked	слева-направо
1	**Унарные**	+ - ! ~ ++x --x	слева-направо
1	**Унарные**	(T)x	**справа-налево**
2	**Мультипликативные**	* / %	слева-направо
3	**Аддитивные**	+ -	слева-направо
4	**Сдвиг**	<< >>	слева-направо
5	**Сравнение, проверка типов**	< > <= >= is as	слева-направо
6	**Равенства**	== !=	слева-направо
7	**Логическое И**	&	слева-направо
8	**Логическое исключающее ИЛИ**	^	слева-направо
9	**Логическое ИЛИ**	|	слева-направо
10	**Условное И**	&&	слева-направо
11	**Условное ИЛИ**	||	слева-направо
12	**Условный ?**	? :	**справа-налево**
13	**Присваивание**	= *= /= %= += -= <<= >>= &= ^= |=	**справа-налево**

Объявления и инициализация

- Для ввода в программу новой переменной – её надо *объявить*
  - *уведомить компилятор об элементе и его типе*

    int x;
- Перед использованием переменные нужно *инициализировать*

    x = 5;
- Правила инициализации:
  - поля структур и классов автоматически обнуляются при создании объекта, если не проинициализированы явно
    - типы значения в 0, ссылочные типы в null
  - локальные переменные запрещено использовать в правой части выражений без предварительной инициализации
    - правило definite assignment

Арность операторов

- Унарный
  - применяется к одному операнду

  `-x,  !b,  (int)d,  ++x`
- Бинарный
  - применяется к двум операндам
  - большинство операторов – бинарные

  `a+b,  b&&c`
- Тернарный
  - применяется к трём операндам
  - единственный тернарный оператор – условный ?

  `(x>y)?x:y`

Левая и правая ассоциативность

- Операторы с одинаковым приоритетом вычисляются на основе *ассоциативности*
  - левая ассоциативность – слева направо
  - правая ассоциативность – справа налево
  - порядок можно изменить с помощью скобок!
- *Присваивание – право ассоциативно*
  - сперва вычисляется правый аргумент

    a = b = c; равносильно a = (b = c);

    (a = b) = c; приводит к ошибке

Блоки и видимость переменных

`Оператор1; Оператор2; … ОператорN;`

– последовательность операторов

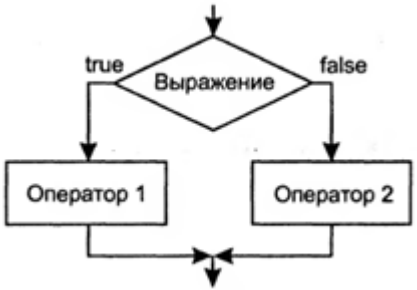
`{ Оператор1; Оператор2; … ОператорN; }`

– блок

- **Локальная переменная** видима внутри блока операторов или метода, в котором она объявлена

Условные операторы

Пример: *В переменную **y** записать значение* **|*x*|+1**

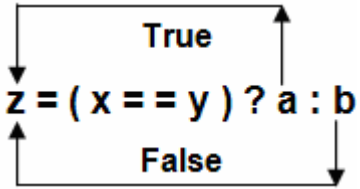


Полный условный оператор:

```
if(x>=0)
    y = x+1;
else
    y = -x+1;
```

Неполный условный оператор:

```
if(x<0)
{
    x *= -1;
}
y = x + 1;
```



Тернарный условный ?

```
y = x >= 0 ? x + 1 : -x + 1;
```

или даже

```
y = 1 + (x >= 0 ? x : -x);
```

Условие

- Выражение типа bool, принимающее аргументы разных типов
  - if (x>0)
    - x – аргумент, (x>0) - значение
  - предикат (мат. логика)
    - в чём разница с булевой функцией?
- В условных и циклических операторах – в круглых скобках

Операторные скобки

- Если в ветках условного оператора или теле цикла выполняется только один оператор, операторные скобки можно опустить
  - блок из одного оператора
- *Если операторов несколько – операторные скобки – обязательны!*
- **Лучше всегда писать**

Dangling else

- При отсутствии операторных скобок else приклеивается к ближайшему if

```
if (условие1)
    if (условие2)
        Команда1;
else
    Команда2;
```

- else будет приклеен ко второму if
  - C# - не Python, на отступы не смотрит

Оператор switch и сложные ветвления

```
int day = 5;

if (day == 1)
    Console.WriteLine("Понедельник");
else if (day==2)
    Console.WriteLine("Вторник");
else if (day==3)
    Console.WriteLine("Среда");
else if (day==4)
    Console.WriteLine("Четверг");
else if (day==5)
    Console.WriteLine("Пятница");
else if (day==6)
    Console.WriteLine("Суббота");
else if (day==7)
    Console.WriteLine("Воскресенье");
else
    Console.WriteLine("Недопустимый номер");
```

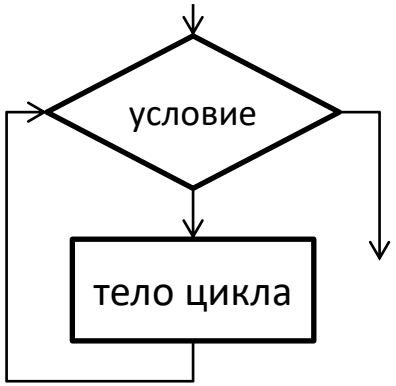
```
int day = 5;
switch (day)
{
    case 1:
        Console.WriteLine("Понедельник");
        break;
    case 2:
        Console.WriteLine("Вторник");
        break;
    case 3:
        Console.WriteLine("Среда");
        break;
    case 4:
        Console.WriteLine("Четверг");
        break;
    case 5:
        Console.WriteLine("Пятница");
        break;
    case 6:
        Console.WriteLine("Суббота");
        break;
    case 7:
        Console.WriteLine("Воскресенье");
        break;
    default:
        Console.WriteLine("Недопустимый номер");
}
```

Switch fallthrough

- В C# в операторе switch запрещено «проваливание» (fallthrough)
- После каждого варианта пишется break;
- Если нужно объединить варианты:

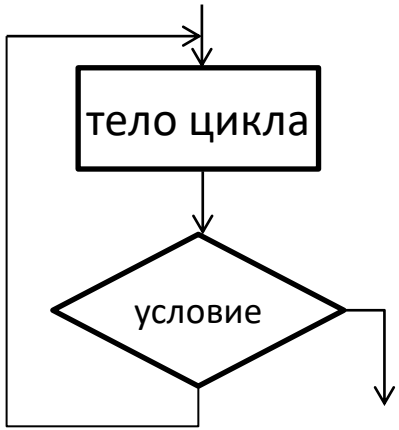
```
switch (/*...*/)
{
    case 0: // shares the exact same code as case 1
    case 1:
        // do something
        goto case 2;
    case 2:
        // do something else
        goto default;
    default:
        // do something entirely different
        break;
}
```

Циклы



- **С предусловием**
  - сперва проверка, потом действия

```
int k = 0;
while(k==0)
{
        int tmp = SomeMethod();
        k = AnotherMethod(tmp);
}
```



- **С постусловием**

сперва действия, потом проверка

```
int k;
do
{
        int tmp = SomeMethod();
        k = AnotherMethod(tmp);
} while (k==0);
```

Пред и постусловие

- Цикл с предусловием (while) функционально эквивалентен циклу с постусловием (do-while)
  - один можно переписать в другой и наоборот
- Упражнение: перепишите в while

```
do
{
    p1; p2; p3;
} while (b);
```

Частный случай while

- Часто while имеет вид:

```
инициализация (для проверки условия)
while (условие)
{
    Последовательность команд;
    Команда перехода на следующую итерацию (переход)
}
```

- Для сокращения записи таких циклов в С-подобных языках используется цикл **for**

Цикл for

- Цикл **for** (с параметром)
  - инициализация, условие, итерация

```
for (int i = 0; i < 10; i++)
      Console.Write('A');
```

или даже так (ненормальный цикл):

```
for (int k, tmp = SomeMethod(); k == 0; k = SomeMethod(tmp)){}
```

Больше «хитрых» циклов

- Цикл foreach по коллекции элементов
  - в цикле перебираются все элементы

```
foreach (var VARIABLE in COLLECTION)
{
        SomeAction();
}
```

Операторы перехода

- **goto** – безусловный переход по метке
- **break** – немедленный выход из цикла, продолжение выполнения со следующего за циклом оператора
- **continue** – немедленное завершение итерации цикла
  - while|do-while – переход к условию
  - for – вычисление итерационного выражения, затем переход к условию
- **Лучше не использовать совсем!**
  - всегда можно заменить на использование логических флажков и проверок

Пустой блок и пустой оператор

- Одинаковое поведение:
  - Если блок/оператор пустой, управление передаётся в конечную точку блока/оператора
- Оба могут использоваться для указания пустого тела цикла/условного оператора
- Пустой оператор может использоваться для создания goto метки перед закрывающей “}” блока

```
{} ;
```

```
while (ProcessMessage())
    ;
while (ProcessMessage())
{}
```

```
void F()
{
    if (done)
        goto exit;
    ...
    exit:;
}
```

Beware of empty statements!

- Не нужно ставить ; после заголовков условных и циклических операторов

```
if (x < 0);
    x = -x;
```

- Это работающий код.
  - который умножает x на -1. Всегда.

Ввод / вывод

- Класс Console
  из пространства имён System
  - стандартные потоки для консольных приложений
    - входной *stdin* (`Console.In`)
    - выходной *stdout* (`Console.Out`)
    - сообщения об ошибках *stderr* (`Console.Error`)

Консольный вывод

- Записывает в стандартный выходной поток строку
  - **`Console.Write(str);`**
    - без переноса строки
  - **`Console.WriteLine(str);`**
    - с переносом строки
- При записи любой переменной в выходной поток, её значение предварительно преобразуется к строке
  - автоматически (для любого типа определено преобразование к строке ToString())
  - !можно переопределить это преобразование для пользовательских типов

Консольный ввод

- ***Считываются символы и строки***
- Считывание символа из стандартного входного потока
  - **`int code=Console.Read();`**
    - считается код символа. Если нужен символ можно явно преобразовать **`(char) code`**
- Считывание строки из стандартного входного потока
  - **`string str=Console.ReadLine();`**
- Если считываются значения не строковых типов в строковом представлении (пример: цифры числа), ***нужно преобразовать строку в нужный тип*** (распарсить)
  - **`int num = int.Parse(str);`** или **`int num = Convert.ToInt32(str);`**
  - или даже **`int num; int.TryParse(str,out num);`**

Вопросы?

e-mail: marchenko@it.kfu.ru

© Марченко Антон Александрович 2016 г.