Лабораторная работа

4. Циклы и функции

Варианты заданий

Выполнить задания 1-4 по предложенным вариантам:

	№ Задания			
	1	2	3	4
Вариант 1	1.I	2.II	3.III	4.IV
Вариант 2	1.II	2.III	3.II	4.V
Вариант 3	1.III	2.I	3.I	4.II
Вариант 4	1.V	2. IV	3. II	4. I
Вариант 5	1.IV	2.V	3.I	4.II
Вариант 6	1.I	2.V	3.III	4.IV
Вариант 7	1.V	2.IV	3.IV	4.II
Вариант 8	1.III	2.III	3.V	4.III
Вариант 9	1.IV	2.I	3.III	4.III
Вариант 10	1.II	2.II	3.V	4.V

Задание 1.

I Дано натуральное число n. Напишите процедуру **static void Squares (int n)**, которая выводит на консоль все точные квадраты натуральных чисел, не превосходящие данного числа n.
(Например, при вводе 50 программа должна вывести 1 4 9 16 25 36 49).

II Дано натуральное число n. Напишите функцию **static int SumOfDigits (int n)**, вычисляющую сумму цифр числа n. Выведите сумму цифр числа n.

III Дано натуральное число n. Напишите функцию **static int NumberOfZeroes (int n)**, определяющую количество нулей среди всех цифр числа n. Выведите результат.

IV Дано натуральное число n. Напишите процедуру **static void Factorials (int n)**, которая выводит на консоль факториалы всех натуральных чисел от 1 до n.

V Дано натуральное число n. Напишите процедуру **static void Sums (int n)**, которая выводит на консоль для каждого числа i (от 1 до n) значение суммы i=1+2+…+i.

Задание 2.

I Вводится последовательность целых чисел до тех пор, пока не будет введено число 0.

После ввода числа 0 программа должна вывести минимальное число.

II Вводится последовательность целых чисел до тех пор, пока не будет введено число 0.

После ввода числа 0 программа должна вывести сумму введенных чисел.

III Вводится последовательность целых чисел до тех пор, пока не будет введено число 0.

После ввода числа 0 программа должна вывести количество отрицательных чисел.

IV Вводится последовательность целых чисел до тех пор, пока не будет введено число 0.

После ввода числа 0 программа должна вывести количество нечетных чисел.

V Вводится последовательность целых чисел до тех пор, пока не будет введено число 0.

После ввода числа 0 программа должна вывести максимальное отрицательное число.

Задание 3.

I Дано натуральное число n. Напишите функцию **static int MaxDigit (int n)**, определяющую наибольшую цифру данного числа (например, при вводе 179 программа выводит 9). Определите время работы функции для 1 000 000 повторов.

II Дано натуральное число n. Напишите функцию **static int NumberOfOdds (int n)**, определяющую количество нечетных цифр среди всех цифр числа n. Выведите результат. Определите время работы функции для 1 000 000 повторов

III Напишите функцию **static int Reverse (int n)**, которая по данному натуральному n возвращает число, составленное из его цифр в обратном порядке (например, Reverse(17962)=26971).
Определите время работы функции для 1 000 000 повторов.

IV Напишите функцию **static int Cut (int n)**, которая по данному натуральному числу n возвращает число, полученное отбрасыванием старшего и младшего разрядов (например, при вводе 179 программа выводит 7). Определите время работы функции для 1 000 000 повторов.

V Напишите функцию **static int UpToTen (int n)**, которая по данному натуральному числу n возвращает число, полученное из n следующим образом: для каждой цифры i числа n в новом числе ставится цифра (10-i), цифра 0 остается без изменений (например,

при вводе 1079 программа выводит 9031). Определите время работы функции для 1 000 000 повторов.

Задание 4.

I Напишите рекурсивную функцию возведения в степень, пользующуюся следующим свойством: $a^n=a*a^{n-1}$

II Напишите рекурсивную функцию возведения в отрицательную степень n: $a^{-n}=1/a^n$

III Для биномиальных коэффициентов (числа сочетаний из n по k) хорошо известна рекуррентная формула: $C_n^k=C_{n-1}^{k-1}+C_{n-1}^k$. Вычислите рекурсивно значение C_n^k, пользуясь этой формулой и учитывая, что $C_n^0=C_n^n=1$.

IV Напишите рекурсивную функцию суммы всех двузначных чисел.

V Напишите рекурсивную функцию для определения является ли натуральное число простым.

Методические указания

Оглавление

4.1. Процедуры и функции - функциональные модули

В языке C# нет специальных ключевых слов - method, procedure, function, но сами понятия, конечно же, присутствуют. Синтаксис объявления метода позволяет однозначно определить, чем является **метод** – процедурой или функцией.

Функция отличается от процедуры двумя особенностями:

- всегда вычисляет некоторое значение, возвращаемое в качестве результата функции (int function() {return 1;});
- вызывается в выражениях (a=funtction();).

Процедура C# имеет свои особенности:

- возвращает формальный результат void, который указывает на отсутствие результата, возвращаемого при вызове процедуры;
- вызов процедуры является оператором языка;
- имеет входные и выходные аргументы, причем выходных аргументов - ее результатов - может быть достаточно много.

Когда компилятор находит в основном тексте программы имя метода, то происходит приостановка выполнения текущего кода программы и осуществляется переход к найденной функции. Когда метод выполнится и завершит свою работу, то произойдет возврат в основной код программы, на ту инструкцию, которая следует за именем метода. Следующий пример иллюстрирует безусловный переход с использованием процедуры:

```
static void Main()
{
    Console.WriteLine("Метод Main. Вызываем метод Jump...");
    Jump();
    Console.WriteLine("Возврат в метод Main.");
}
static void Jump()
{
    Console.WriteLine("Работает метод Jump!");
}
```

Результат работы:

```
Метод Main. Вызываем метод Jump...
Работает метод Jump!
Возврат в метод Main.
```

Программа начинает выполняться с метода Main() и осуществляется последовательно, пока компилятор не вызовет процедуру Jump(). Таким образом, происходит ответвление от основного потока выполняемых инструкций. Когда процедура Jump() заканчивает свою работу, то продолжается выполнение программы со следующей строки после вызова процедуры.

4.1.1. Описание методов (процедур и функций). Синтаксис

Синтаксически в описании метода различают две части - описание заголовка и описание тела метода:

```
заголовок_метода
{
```

```
        тело_метода
}
```

Рассмотрим синтаксис заголовка метода:

```
[атрибуты][модификаторы]{void | тип_результата_функции}
имя_метода([список_формальных_аргументов])
```

Имя метода и список формальных аргументов составляют сигнатуру метода. Квадратные скобки (метасимволы синтаксической формулы) показывают, что атрибуты и модификаторы могут быть опущены при описании метода. Заметьте, в сигнатуру не входят имена формальных аргументов, здесь важны типы аргументов. В сигнатуру не входит и тип возвращаемого результата.

Обязательным при описании заголовка является указание типа результата, имени метода и круглых скобок, наличие которых необходимо и в том случае, если сам список формальных аргументов отсутствует. Формально тип результата метода указывается всегда, но значение void однозначно определяет, что метод реализуется процедурой. Тип результата, отличный от void, указывает на функцию.

Пример простейшего метода:

```
static int GetSum(int x, int y)
   {
       return x+y;
   }
```

Тип определяет, значение какого типа вычисляется с помощью метода. Часто употребляется термин "**метод возвращает значение**". Если метод не возвращает никакого значения, в его заголовке задается тип void, а оператор return отсутствует.

Параметры используются для обмена информацией с методом. Параметр представляет собой локальную переменную, которая при вызове метода принимает значение соответствующего аргумента. Область действия параметра — весь метод.

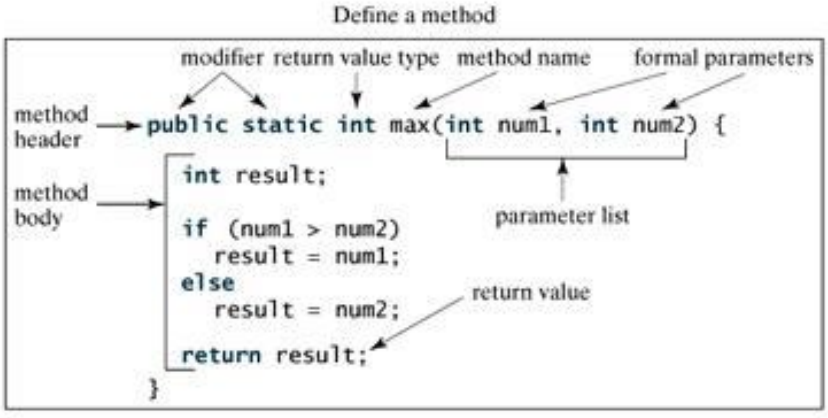

Рис. 1. Основные понятия, связанные с методами

Например, чтобы вычислить значение синуса для вещественной величины **x**, мы передаем ее в качестве аргумента в метод Sin класса Math, а чтобы вывести значение этой переменной на экран, мы передаем ее в метод WriteLine класса Console:

```
double x = 0.1;
double y = Math.Sin(x);
Console.WriteLine(x);
```

При этом метод Sin возвращает в точку своего вызова вещественное значение синуса, которое присваивается переменной y, а метод WriteLine ничего не возвращает. Иллюстрация вызова метода приведена на рис. 2.

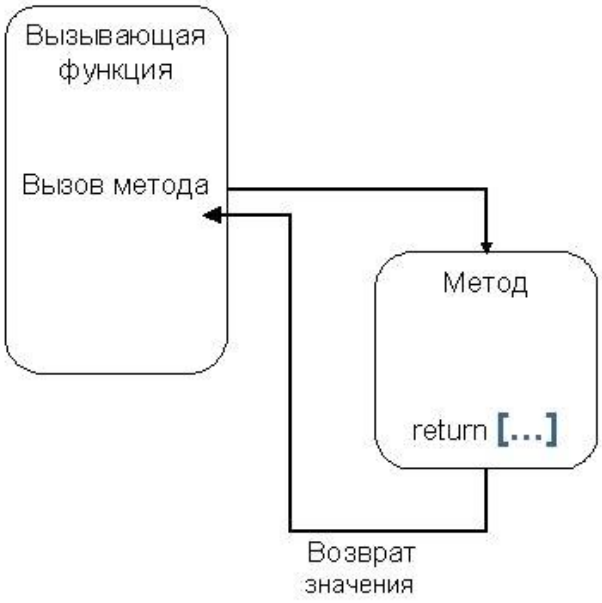

Рис. 2. Вызов метода

Синтаксически тело метода является блоком, который представляет собой последовательность операторов и описаний переменных, заключенную в фигурные скобки. Если речь идет о теле функции, то в блоке должен быть хотя бы один оператор, возвращающий значение функции в форме return <выражение>.

Переменные, описанные в блоке, считаются локализованными в этом блоке. В записи операторов блока участвуют имена локальных переменных блока, имена полей класса и имена аргументов метода.

Метод, не возвращающий значение, вызывается отдельным оператором, а метод, возвращающий значение, — в составе выражения в правой части оператора присваивания.

Параметры, описываемые в заголовке метода, определяют множество значений аргументов, которые можно передавать в метод. Список аргументов при вызове как бы накладывается на список параметров, поэтому они должны попарно соответствовать друг другу. Для каждого параметра должны задаваться его тип и имя. Например, заголовок метода Sin выглядит следующим образом:

```
public static double Sin( double a );
```

Имя метода вкупе с количеством, типами и спецификаторами его параметров представляет собой сигнатуру метода. В классе не должно быть методов с одинаковыми сигнатурами. В современном представлении сигнатура метода совпадает с его заголовком:

```
public static int GetSum(int x, int y);
```

Краткая запись (классическое представление сигнутуры) – не содержит модификаторов, типа возвращаемого параметра и имен переменных:

```
GetSum(int, int);
```

4.1.2. Вызов метода. Синтаксис и семантика

Сам вызов метода, независимо от того, процедура это или функция, имеет один и тот же синтаксис:

```
имя_метода([список_фактических_аргументов])
```

Что происходит в момент вызова метода? Выполнение начинается с вычисления фактических аргументов, которые, как мы знаем, являются выражениями. Вычисление этих выражений может приводить, в свою очередь, к вызову других методов, так что этот первый этап может быть довольно сложным и требовать больших временных затрат.

Если бы синтаксис описания метода допускал отсутствие скобок у функции (метода) в случае, когда список аргументов отсутствует, то клиент класса мог бы и не знать, обращается он к полю или к методу. Когда мы хотим получить длину строки, то пишем s.Length, точно зная, что Length - это поле, а не метод класса string. Если бы по каким-либо причинам разработчики класса string решили изменить реализацию и заменить поле Length соответствующей функцией, то ее вызов имел бы вид s.Length().

4.1.3. Параметры методов

При вызове метода выполняются следующие действия:

1. Вычисляются выражения, стоящие на месте аргументов.
2. Выделяется память под параметры метода в соответствии с их типом.
3. Каждому из параметров сопоставляется соответствующий аргумент (аргументы как бы накладываются на параметры и замещают их).
4. Выполняется тело метода.

Если метод возвращает значение, оно передается в точку вызова; если метод имеет тип void, управление передается на оператор, следующий после вызова.

При этом проверяется соответствие типов аргументов и параметров и при необходимости выполняется их преобразование. При несоответствии типов выдается диагностическое сообщение.

```
static int Max(int a, int b)  // метод выбора максимального значения
{
    if (a > b) return a;
    else return b;
}
static void Main(){
    int a = 2, b = 4;
    int x = Max(a, b);                // вызов метода Max
    Console.WriteLine(x);             // результат: 4

    short t1 = 3, t2 = 4;
    int y = Max(t1, t2);              // вызов метода Max
    Console.WriteLine(y);             // результат: 4

    int z = Max(a + t1, t1 / 2 * b);  // вызов метода Max
    Console.WriteLine(z);             // результат: 5
}
```

Главное требование при передаче параметров состоит в том, что аргументы при вызове метода должны записываться в том же порядке, что и в заголовке метода, и должно существовать неявное преобразование типа каждого аргумента к типу соответствующего параметра. Количество аргументов должно соответствовать количеству параметров. Иллюстрация приведена на рис. 3.

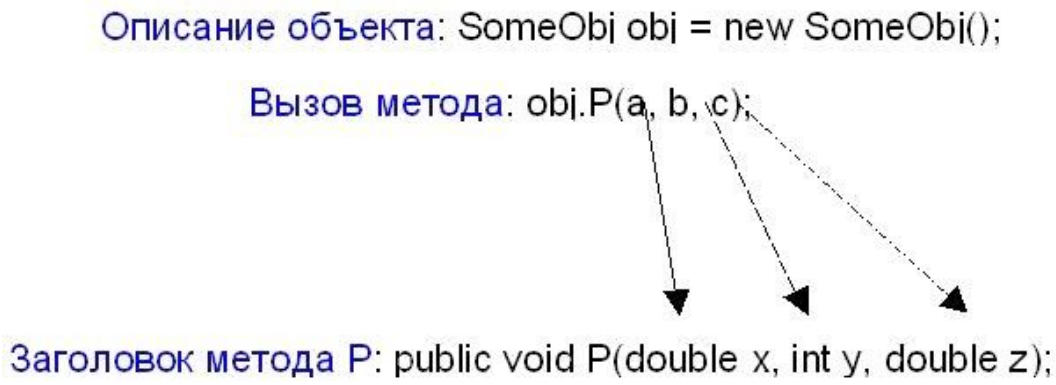

Рис. 3. Соответствие параметров при вызове метода

Существуют два способа передачи параметров: по значению и по ссылке.

*При передаче **по значению*** метод получает копии значений аргументов, и операторы метода работают с этими копиями. Доступа к исходным значениям аргументов у метода нет, а, следовательно, нет и возможности их изменить.

*При передаче **по ссылке*** (*по адресу*) метод получает копии адресов аргументов, он осуществляет доступ к ячейкам памяти по этим адресам и может изменять исходные значения аргументов, модифицируя параметры.

В C# для обмена данными между вызывающей и вызываемой функциями предусмотрено четыре типа параметров:

- параметры-значения;
- параметры-ссылки — описываются с помощью ключевого слова **ref** ;
- выходные параметры — описываются с помощью ключевого слова **out** ;
- параметры-массивы — описываются с помощью ключевого слова **params**.

Ключевое слово предшествует описанию типа параметра. Если оно опущено, параметр считается параметром-значением. Параметр-массив может быть только один и должен располагаться последним в списке, например:

```
public int Calculate( int a, ref int b, out int c, params int[] d ) …
```

4.1.4. Параметры-значения

Параметр-значение описывается в заголовке метода следующим образом:

```
тип имя
```

Пример заголовка метода, имеющего один параметр-значение целого типа:

```
void P( int x )
```

Имя параметра может быть произвольным. Параметр x представляет собой локальную переменную, которая получает свое значение из вызывающей функции при вызове метода. В метод передается копия значения аргумента.

Механизм передачи следующий: из ячейки памяти, в которой хранится переменная, передаваемая в метод, берется ее значение и копируется в специальную область памяти — область параметров. Метод работает с этой копией. По завершении работы метода область параметров освобождается. Этот способ годится только для передачи в метод исходных данных.

При вызове метода на месте параметра, передаваемого по значению, может находиться выражение, для типа которого существует неявное преобразование типа выражения к типу параметра.

Например, пусть в вызывающей функции описаны переменные и им до вызова метода присвоены значения:

```
int    x = 1;
sbyte  c = 1;
ushort y = 1;
```

Тогда следующие вызовы метода P, заголовок которого был описан ранее, будут синтаксически правильными:

```
P( x );    P( c );    P( y );    P( 200 );    P( x / 4 + 1 );
```

4.1.5. Параметры-ссылки

Признаком параметра-ссылки является ключевое слово ref перед описанием параметра:

```
ref тип имя
```

Пример заголовка метода, имеющего один параметр-ссылку целого типа:

```
void P( ref int x )
```

При вызове метода в область параметров копируется адрес аргумента, и метод через него имеет доступ к ячейке, в которой хранится аргумент. Метод работает непосредственно с переменной из вызывающей функции и, следовательно, может ее изменить, поэтому если в методе требуется изменить значения параметров, они должны передаваться только по ссылке.

При вызове метода на месте параметра-ссылки может находиться только ссылка на инициализированную переменную точно того же типа. Перед именем параметра указывается ключевое слово ref.

Проиллюстрируем передачу параметров-значений и параметров-ссылок на примере

```
static void P(int a, ref int b)
{
    a = 44; b = 33;
    Console.WriteLine("внутри метода {0} {1}", a, b);
}
static void Main()
{
    int a = 2, b = 4;
    Console.WriteLine("до вызова     {0} {1}", a, b);
    P(a, ref b);
    Console.WriteLine("после вызова  {0} {1}", a, b);
}
```

Результаты работы этой программы:

```
до вызова     2 4
внутри метода 44 33
после вызова  2 33
```

Несколько иная картина получится, если передавать в метод не величины значимых типов, а *экземпляры классов*, то есть величины ссылочных типов. Для простоты можно считать, что объекты всегда передаются по ссылке.

4.1.6. Выходные параметры

Довольно часто возникает необходимость в методах, которые формируют несколько величин. В этом случае становится неудобным ограничение параметров-ссылок: необходимость присваивания значения аргументу до вызова метода. Это ограничение снимает спецификатор out. Параметру, имеющему этот спецификатор, должно быть обязательно присвоено значение внутри метода.

Изменим описание второго параметра так, чтобы он стал выходным.

```
static void P(int a, out int b)
{
    a = 44; b = 33;
    Console.WriteLine("внутри метода {0} {1}", a, b);
}
static void Main()
{
    int a = 2, b;
    P(a, out b);
    Console.WriteLine("после вызова  {0} {1}", a, b);
}
```

При вызове метода перед соответствующим параметром тоже указывается ключевое слово out.

4.2. Рекурсия

Рекурсия является одним из наиболее мощных средств в арсенале программиста. Рекурсивные структуры данных и рекурсивные методы широко используются при построении программных систем. Рекурсивные методы, как правило, наиболее всего удобны при работе с рекурсивными структурами данных - списками, деревьями. Рекурсивные методы обхода деревьев служат классическим примером.

Метод P (процедура или функция) называется *рекурсивным*, если при выполнении тела метода происходит вызов самого себя (метода P).

Рекурсия может быть прямой, если вызов P происходит непосредственно в теле этого же метода P. Рекурсия может быть косвенной, если в теле P вызывается метод Q (эта цепочка может быть продолжена), в теле которого опять вызывается метод P. Определения методов P и Q взаимно рекурсивны, если в теле метода Q вызывается метод P, вызывающий, в свою очередь, метод Q.

Для того чтобы рекурсия не приводила к зацикливанию, в тело рекурсивного метода всегда встраивается оператор выбора, одна из ветвей которого не содержит рекурсивных вызовов. Если в теле рекурсивного метода рекурсивный вызов встречается только один раз, значит, что рекурсию можно заменить обычным циклом, что приводит к более

эффективной программе, поскольку реализация рекурсии требует временных затрат и работы со стековой памятью. Вначале рассмотрим простейший пример рекурсивного определения функции, вычисляющей факториал целого числа:

```
    static long factorial(int n)
    {
        if (n <= 1) return (1);
        else return (n * factorial(n - 1));
    }
```

Функция factorial является примером прямого рекурсивного определения – в ее теле она сама себя вызывает. Здесь, как и положено, есть нерекурсивная ветвь, завершающая вычисления, когда n становится равным единице. Это пример так называемой "хвостовой" рекурсии, когда в теле встречается ровно один рекурсивный вызов, стоящий в конце соответствующего выражения. Хвостовую рекурсию намного проще записать в виде обычного цикла. Вот циклическое (не рекурсивное) определение той же функции:

```
    static long fact(int n)
    {
        long res = 1;
        for (int i = 2; i <= n; i++) res *= i;
        return (res);
    }
```

Конечно, циклическое определение проще, понятнее и эффективнее, и применять рекурсию в подобных ситуациях не следует. Интересно сравнить время вычислений, дающее некоторое представление о том, насколько эффективно реализуется рекурсия. Вот соответствующий тест, решающий эту задачу:

```
    static void TestTailRec()
    {
        long time1, time2;
        long f = 0;
        time1 = getTimeInMilliseconds();
        for (int i = 1; i < 1000000; i++) f = fact(15);
        time2 = getTimeInMilliseconds();
        Console.WriteLine(" f= {0}, " + "Время работы циклической процедуры:
{1}", f, time2 - time1);
        time1 = getTimeInMilliseconds();
        for (int i = 1; i < 1000000; i++) f = factorial(15);
        time2 = getTimeInMilliseconds();
        Console.WriteLine(" f= {0}, " + "Время работы рекурсивной процедуры:
{1}", f, time2 - time1);
    }
```

Каждая из функций вызывается в цикле, работающем 1000000 раз. До начала цикла и после его окончания вычисляется текущее время. Разность этих времен и дает оценку времени работы функций. Обе функции вычисляют факториал числа 15.

Проводить сравнение эффективности работы различных вариантов - это частый прием, используемый при разработке программ. Встроенный тип DateTime обеспечивает необходимую поддержку для получения текущего времени. Он совершенно необходим, когда приходится работать с датами. Рассмотрим на примере функции для получения текущего времени, измеряемого в миллисекундах. Статический метод Now класса DateTime возвращает объект этого класса, соответствующий дате и времени в момент

создания объекта. Многочисленные свойства этого объекта позволяют извлечь требуемые характеристики. Текст функции getTimeInMilliseconds:

```
static long getTimeInMilliseconds()
{
    DateTime time = DateTime.Now;
    return (((time.Hour * 60 + time.Minute) * 60 + time.Second) * 1000
        + time.Millisecond);
}
```

Результаты измерений времени работы рекурсивного и циклического вариантов функций слегка отличаются от запуска к запуску, но порядок остается одним и тем же. Эти результаты показаны на рис. 4.

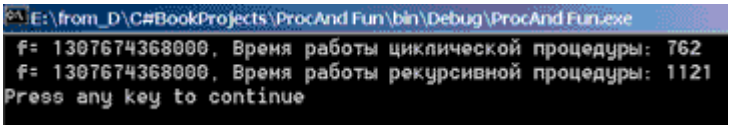

Рис. 4. Сравнение времени работы циклической и рекурсивной функций

Любой нерекурсивный метод можно реализовать без применения рекурсии.

К достоинствам рекурсии можно отнести компактность записи. К недостаткам - расход времени и памяти на повторные вызовы метода и передачи ему копий параметров. Опасность переполнения стека.

Источники:

1. Биллиг В. Основы программирования на C#. Режим доступа: http://www.intuit.ru/studies/courses/2247/18/info
2. Павловская Т.А. Программирование на C#. Учебный курс. Режим доступа: http://ips.ifmo.ru/courses/csharp/index.html
3. Изучаем C Sharp (C#). Программирование на C Sharp (C#) с нуля. Режим доступа: http://simple-cs.ru/