# Функции (методы)

## Понятие функции

С увеличением объема программы становится невозможно удерживать в памяти все детали. Чтобы уменьшить сложность программы, ее разбивают на части (декомпозиция – метод борьбы со сложностью). В C# задача может быть разделена на более простые подзадачи с помощью функций.

**Функция** – это именованная последовательность описаний и операторов, выполняющая законченное действие, например, формирование массива, печать массива и т. д. Любая функция содержит одну или несколько инструкций (операторов), которые называют телом функции.

**Преимущества использования функций**

1. Разделение задачи на функции позволяет сократить сложность задачи.
2. Разделение задачи на функции позволяет избежать избыточности кода, т. к. функцию записывают один раз, а вызывают многократно.
3. Можно использовать одну функцию в разных задачах, т.е. сокращается время работы программиста и повышается качество работы, т.к. эта функция уже будет работать без ошибок.
4. Программу, которая содержит функции, легче отлаживать, т.к можно сначала протестировать каждую функцию отдельно (простые задачи), а затем уже интегрировать их в сложную задачу.

В хорошей программе одна функция выполняет только **одну** задачу. Каждая функция имеет имя, которое используется для ее вызова. В общем случае функции можно присвоить любое имя. Но имя Main () зарезервировано для функции, с которой начинается выполнение программы. Кроме того, в качестве имен функций нельзя использовать ключевые слова С#.



Рисунок 1 Функция как минимальный исполняемый модуль программы

Упрощенный формат записи функции следующий:

тип имя\_функции([список\_формальных параметров])

{

тело\_функции

}

Тело\_функции – это блок или составной оператор. Внутри функции нельзя **определить** другую функцию.

В теле функции должен быть оператор, который **возвращает** полученное значение функции в точку вызова. Он может иметь две формы:

1. return выражение;
2. return;

Первая форма используется для **возврата результата**, поэтому выражение должно иметь тот же тип, что и тип функции в определении. Вторая форма используется, если функция **не возвращает значения**, т. е. имеет тип void. Программист может не использовать этот оператор в теле функции явно, компилятор добавит его автоматически в конец функции перед }. Это может быть любой допустимый тип, включая типы классов, создаваемые программистом.

**Список формальных параметров** – это те величины, которые требуется передать в функцию. Элементы списка разделяются запятыми. Для каждого параметра указывается тип и имя.

Для того, чтобы выполнялись операторы, записанные в теле функции, функцию необходимо вызвать. При вызове указываются: имя функции и фактические параметры. **Фактические параметры** заменяют формальные параметры при выполнении операторов тела функции. Фактические и формальные параметры должны совпадать по количеству и типу.

В.В. Подбельский предлагает разделять методы-процедуры и методы-функции. **Метод-процедура** отличается от метода-функции следующими свойствами:

1. В качестве типа возвращающего значения используется тип void. Т.е. в точку вызова не возвращается никакого результата.

2. В теле процедуры отсутствует оператор return. Если этот оператор присутствует, то он не содержит выражения для вычисления возвращаемого значения.

3. Для обращения к методу-процедуре используется только оператор вызова метода.

имя\_метода (список аргументов);

**Метод-функция:**

1. В качестве типа возвращаемого значения используется ссылочный или значимый тип. Т.е. функция всегда возвращает результат в точку вызова.
2. В теле функции всегда присутствует хотя бы один оператор возврата:

return выражение;

выражение и определяет возвращаемый результат.

1. Результатом метода-функции всегда является **одно** значение.
2. Обращение к методу-функции может использоваться как операнд выражения (должно быть согласование по типу).

результат= имя\_метода (список аргументов);

1. Если результат не используется в выражении, то функцию можно вызвать как отдельный оператор (как метод-процедуру).

Мы будем рассматривать только методы классов, поэтому у каждого метода должен быть модификатор static.

**Пример.** Заданы координаты сторон треугольника, если такой треугольник существует, то найти его площадь.

**Анализ задачи**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Описание функции** | **Входные данные** | **Классы входных данных** | **Выходные данные** |
| 1 | Определяет длину отрезка, заданного координатами (x1,y1), (х2,у2) (Line) | double x1,double y1,double x2,double y2 | Вещественные числа,  целые числа | Вещественное число |
| 2 | Определяет площадь треугольника, заданного длинами сторон а,b,c (Square) | double a, double b, double c | Вещественные числа,  целые числа | Вещественное число |
| 3 | Определяет существует ли треугольник (Triangle) | double a, double b, double c | Вещественные числа,  целые числа | true/false |
| 4 | Вводит координаты точки | int number – номер точки | Вещественные числа,  целые числа | double x,  double y |
| Cимволы | Сообщение об ошибке  x=0  y=0 |

class Program

{

/\*функция возвращает длину отрезка, заданного координатами x1,y1 и x2,y2\*/

static double Line(double x1, double y1, double x2, double y2)

{

return Math.Sqrt(Math.Pow(x1 - x2, 2) + Math.Pow(y1 - y2, 2));

}

/\*функция возвращает площадь треугольника, заданного длинами сторон а,b,c\*/

static double Square(double a, double b, double c)

{

double s, p = (a + b + c) / 2;

return s = Math.Sqrt(p \* (p - a) \* (p - b) \* (p - c));//формула Герона

}

//возвращает true, если треугольник существует

static bool Triangle(double a, double b, double c)

{

if (a + b > c && a + c > b && c + b > a) return true;

else return false;

}

static void MakePoint(int number, out double x, out double y)

{

try

{

Console.WriteLine("Введите координаты {0} точки:", number);

Console.Write("Первая координата:");

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

Console.Write("Вторая координата:");

y = Convert.ToDouble(Console.ReadLine());

}

catch (FormatException)

{

Console.WriteLine("Ошибка при вводе координат {0} точки!", number);

x = 0;

y = 0;

}

}

static void Main(string[] args)

{

double x1, y1, x2, y2, x3, y3;

double point1\_2, point1\_3, point2\_3;

do

{

MakePoint(1, out x1, out y1);

MakePoint(2, out x2, out y2);

MakePoint(3, out x3, out y3);

point1\_2 = Line(x1, y1, x2, y2);

point1\_3 = Line(x1, y1, x3, y3);

point2\_3 = Line(x2, y2, x3, y3);

if (Triangle(point1\_2, point1\_3, point2\_3) == true)

Console.WriteLine("S=" + square(point1\_2, point2\_3, point1\_3));

else Console.WriteLine("Треугольник не существует");

}

while (!(x1 == 0 && y1 == 0 && x2 == 0 && y2 == 0 && x3 == 0 && y3 == 0));

}

Когда мы пишем свои методы (или используем уже готовые) важно понимать, откуда метод берет свои данные и куда он будет отправлять результаты работы.

**Исходные данные** могут быть получены:

1. как параметры метода;
2. как глобальные переменные (по отношению к методу);
3. от внешних устройств (файлы, потоки ввода).

**Результаты** метод может передавать:

1. в точку вызова, как возвращаемое функцией значение;
2. в глобальные по отношению к методу объекты (переменные);
3. внешним устройствам (файлы, потоки вывода);
4. через параметры метода.

Обмен через глобальные объекты нарушает один из основополагающих принципов ООП – инкапсуляцию, поэтому в реальных разработках его использовать не рекомендуют (также как goto).

Обмен со стандартными потоками ввода-вывода реализуется с помощью класса Console.

Файловые потоки будут рассмотрены позже.

## 2. Параметры функции

Рассмотрим передачу данных с помощью **механизма параметров**.

Основным способом обмена информацией между вызываемой и вызывающей функциями является механизм параметров.

При определении метода в его заголовке размещается спецификация параметров, она может быть пустой. Спецификация параметров – последовательность описаний параметров:

модификатор тип\_параметра имя\_параметра

Модификатор может отсутствовать или иметь одну из следующих форм: ref, out, params.

Параметр может быть любого типа (базового, строкой, object, массив, перечисление и т.д.).

Имя параметра – идентификатор, выбираемый программистом. Область видимости и время действия параметра – заголовок и тело функции. Т.е. вне кода функции параметры не определены и недоступны.

Существуют следующие способы передачи параметров в функцию:

1. по значению;
2. по ссылке (ref);
3. выходные параметры (out);
4. массив-параметр (params).

При **передаче по значению** выполняются следующие действия:

1. вычисляются значения выражений, стоящие на месте фактических параметров;
2. в стеке выделяется память под формальные параметры функции;
3. каждому фактическому параметру присваивается значение формального параметра, при этом проверяются соответствия типов и при необходимости выполняются их преобразования.

Пример1.

static double square(double a, double b, double c)

{

double s, p=(a+b+c)/2;

return s = Math.Sqrt(p \* (p - a) \* (p - b) \* (p - c));//формула Герона

}

Пример 2.

class Program

{

public static void Change(int a, int b)

{

int r = a;

a = b;

b = r;

}

static void Main(string[] args)

{

int x = 10, y = 5;

Console.WriteLine("x=" + x + " y=" + y);

Change(x,y);

Console.WriteLine("x=" +x + " y=" + y);

}

Таким образом, при передаче параметров по значению в стек заносятся копии фактических параметров, и операторы функции работают с этими копиями. Доступа к самим фактическим параметрам у функции нет, следовательно, нет возможности их изменить.

Чтобы метод мог с помощью параметров изменять внешние по отношению к методу объекты, параметры должны иметь мо­дификатор **ref**, т.е. передаваться **по ссылке**.

class Program

{

public static void Change(ref int a, ref int b)

{

int r = a;

a = b;

b = r;

}

static void Main(string[] args)

{

int x = new int(), y = new int();

x = 5; y = 10;

Console.WriteLine("x=" + x + " y=" + y);

Change(ref x, ref y);

Console.WriteLine("x=" + x + " y=" + y);

}

Параметры, передаваемые по ссылке, используют­ся для изменения уже существующих значений, внешних по отношению к методу объектов.

**Выходные параметры** снабжаются модификатором out и позволяют присвоить значения объектам вызывающего метода даже в тех случаях, когда эти объекты значений еще не имели. Причина, по которой компилятор позволяет передавать на первый взгляд неинициализированные данные, связана с тем, что в вызываемом методе операция присваивания должна выполняться обязательно.

static void MakePoint(int number, out double x, out double y)

{

try

{

Console.WriteLine("Введите координаты {0} точки:", number);

Console.Write("Первая координата:");

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

Console.Write("Вторая координата:");

y = Convert.ToDouble(Console.ReadLine());

}

catch (FormatException)

{

Console.WriteLine("Ошибка при вводе координат {0} точки!", number);

x = 0;

y = 0;

}

}

static void Main(string[] args)

{

double x1, y1, x2, y2, x3, y3;

. . . .

MakePoint(1, out x1, out y1);

MakePoint(2, out x2, out y2);

MakePoint(3, out x3, out y3);

. . .

}

Метод с выходными параметрами (имеющими модифика­торы ref или out) обычно выступает в роли процедуры. Хотя не запрещено такому методу возвращать значение с помощью оператора return.

**Отличия между ссылочными и выходными параметрами.**

• Выходные параметры (out) не нужно инициализировать перед передачей методу, т.к. метод сам должен присваивать значения выходным параметрам перед выходом. Т.е. этих данных не было до выполнения функции и функция их должна создать.

• Ссылочные параметры (ref) нужно обязательно инициализировать перед передачей методу, т.к. они подразумевают передачу ссылки на уже существующую переменную. Т.е. эти данные существовали в памяти компьютера до выполнения функции и функция их должна изменить.

## 3. Функции с переменным числом параметров

В С# поддерживается использование массивов параметров за счет применения ключевого слова params. Ключевое слово params позволяет передавать методу переменное количество аргументов одного типа в виде единственного логического параметра. Аргументы, помеченные ключевым словом params, могут обрабатываться, если вызывающий код на их месте передает строго типизированный массив или разделенный запятыми список элементов. Конечно, это может вызывать путаницу. Рассмотрим функцию, которая бы позволила вызывающему коду передавать любое количество аргументов и возвращала бы их среднее значение.

static double CalcAverage(params double[] values)

{

double avarage = 0;

if (values.Length == 0) return 0;

foreach (double x in values)

avarage += x;

avarage /= values.Length;

return avarage;

}

static void Main(string[] args)

{

//1 способ

Console.WriteLine("Среднее арифметическое="+CalcAverage(0.2, 0.3, 0.5, 0.7));

//2 способ

double[] mas = { 0.2, 0.3, 0.5, 0.7 };

Console.WriteLine("Среднее арифметическое=" + CalcAverage(mas));

//3 способ

Console.WriteLine("Среднее арифметическое=" + CalcAverage());

}

Если бы в определении CalculateAverage () не было модификатора params, первый способ вызова этого метода приводил бы к ошибке на этапе компиляции, т.к. тогда компилятор искал бы версию CalculateAverage(), принимающую пять аргументов double.

С# требует, чтобы в любом методе поддерживался только один аргумент params, который должен быть последним в списке параметров.

## 4. Необязательные параметры

В определении функции может содержаться умалчиваемое значение параметра. Это значение используется, если при вызове функции соответствующий параметр опущен. Все параметры, описанные справа от такого параметра, также должны быть умалчиваемыми.

static void Print(string s = "номер дома", int value = 1)

{

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

}

static void Main(string[] args)

{

Print();

Print("номер квартиры");

Print(,2);//Error!!!

Print("номер квартиры",15);

}

Значение, присваиваемое необязательному параметру, должно быть известно во время компиляции и не может вычисляться во время выполнения.

static void Print(double [] mas,int size=mas.Length)

{

// int size = mas.Length;

for (int i = 0; i < size; i++)

Console.Write(mas[i] + " ");

Console.WriteLine();

}

static void Main(string[] args)

{

double[] mas = { 0.2, 0.3, 0.5, 0.7 };

Print(mas);

}

Error – т.к. значение time вычисляется во время выполнения программы.

## Именованные параметры

Именованные аргументы позволяют вызывать метод с указанием значений параметров в любом желаемом порядке. Следовательно, вместо того, чтобы передавать параметры исключительно в соответствии с позициями, в которых они определены (как приходится поступать в большинстве случаев), можно указывать имя каждого аргумента, двоеточие и конкретное значение. Именованные аргументы должны всегда размещаться в конце вызова метода.

class Program

{

static void DisplayMessage(ConsoleColor textColor, ConsoleColor backgroundColor, string message)

{

// Сохранение старых цветов для обеспечения возможности

//их восстановления сразу после вывода сообщения.

ConsoleColor oldTextColor = Console.ForegroundColor;//свойство - возвращает цвет фона консоли, т.е. цвет символов

ConsoleColor oldbackgroundColor = Console.BackgroundColor;//свойство - возвращает цвет фона

// Установка новых цветов и вывод сообщения.

Console.ForegroundColor = textColor;

Console.BackgroundColor = backgroundColor;

Console.WriteLine(message);

// Восстановление предыдущих цветов.

Console.ForegroundColor = oldTextColor;

Console.BackgroundColor = oldbackgroundColor;

}

static void Main(string[] args)

{

DisplayMessage(message: "Печатаем сообщение ", textColor: ConsoleColor.DarkRed, backgroundColor: ConsoleColor.White);

DisplayMessage(backgroundColor: ConsoleColor.Green, message: "Меняем цвет...", textColor: ConsoleColor.DarkBlue);

DisplayMessage(ConsoleColor.Blue, message: "Testing...", backgroundColor: ConsoleColor.White);

// DisplayMessage(message: "Testing...", backgroundColor: ConsoleColor.White, ConsoleColor.Blue); //Error!

}

}

Эту конструкцию удобно использовать, если в методе есть необязательные аргументы. Необязательные аргументы (см. выше) должны быть последними в списке параметров.

static void DisplayMessage(ConsoleColor textColor=ConsoleColor.Blue, ConsoleColor backgroundColor=ConsoleColor.White, string message="Test...."){...}

...

DisplayMessage(backgroundColor: ConsoleColor.Green);

DisplayMessage(message:"Hello");

## Перегрузка методов

Цель перегрузки состоит в том, чтобы функция с одним именем по-разному выполнялась и возвращала разные значения при обращении к ней с различными типами и различным числом фактических параметров. Для обеспечения перегрузки необходимо для каждой перегруженной функции определить возвращаемые значения и передаваемые параметры так, чтобы каждая перегруженная функция отличалась от другой функции с тем же именем. Компилятор определяет, какую функцию выбрать по типу фактических параметров.

static int MaxValue(int a, int b)

{

if (a > b)

return a;

else

return b;

}

static double MaxValue(double a, double b)

{

if (a > b)

return a;

else

return b;

}

static string MaxValue(string a, string b)

{

if (String.Compare(a,b)>0)

return a;

else

return b;

}

static void Main(string[] args)

{

Console.WriteLine("максимальное значение из {0} и {1} = {2}",1,5, MaxValue(1,5));

Console.WriteLine("максимальное значение из {0} и {1} = {2}", 1.5, 5.1, MaxValue(1.5, 5.1));

Console.WriteLine("максимальное значение из {0} и {1} = {2}", "111","555", MaxValue("111","555"));

}

}

Среда разработки Visual Studio 2010 обеспечивает помощь при вызове перегруженных методов. При вводе имени перегруженного метода (как, например, хорошо знакомого Console .WriteLine ()) в списке предлагаются все его доступные версии.

## Рекурсивные функции

Рекурсией называется ситуация, когда какой-то алгоритм вызывает себя прямо (прямая рекурсия) или через другие алгоритмы (косвенная рекурсия) в качестве вспомогательного. Сам алгоритм называется рекурсивным.

Рекурсивное решение задачи состоит из двух этапов:

1. исходная задача сводится к новой задаче, похожей на исходную, но несколько проще;

2. подобная замена продолжается до тех пор, пока задача не станет тривиальной, т. е. очень простой.

**Задача 1**. Вычислить факториал (n!), используя рекурсию.

*Исходные данные: n*

*Результат: n!*

Рассмотрим эту задачу на примере вычисления факториала для n=5. Более простой задачей является вычисление факториала для n=4. Тогда вычисление факториала для n=5 можно записать следующим образом:

5!=4!\*5.

Аналогично:

4!=3!\*4;

3!=2!\*3;

2!=1!\*2 ;

1!=0!\*1

Тривиальная (простая) задача:

0!=1.

Можно построить следующую *математическую модель*:



class Program

{

static int fact(int n)

{

if (n==0)return 1;//тривиальная задача

return (n\*fact(n-1));

}

static void Main(string[] args)

{

Console.WriteLine("Введите число для вычисления факториала");

int k = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("{0}!={1}",k,fact(k));

}

}

**Задача 2**. Вычислить степень, используя рекурсию.

*Исходные данные: x,n*

*Результат: xn*

*Математическая модель:*



static int pow(int x, int y)

{

if (y == 0) return 1;

else return (x \* pow(x, y - 1));

}

static void Main(string[] args)

{

Console.WriteLine("Введите числа для вычисления степени");

int x = Convert.ToInt32(Console.ReadLine());

int y = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("{0}^{1}={2}", x,y, pow(x,y));

}