# Лекция 3. Циклы. Одномерные массивы

**Что делать?**

1. Повторить теорию из прошлой лекции:
   1. типы данных;
   2. операции присваивания;
   3. условный оператор **if**;
   4. сравнения и логические операции.
2. Ознакомиться с теорией по темам:
   1. цикл с предусловием **while**;
   2. цикл с постусловием **do-while**;
   3. цикл с параметром **for**;
   4. сравнение циклов;
   5. блок-схемы алгоритмов;
   6. одномерные массивы.
3. Реализовать примеры из данного файла. Выполнять коммит тогда, когда это указано в задании, или чаще. Ответить на вопросы в тексте.
4. Выполнить задания на самостоятельную работу. Если выполняется дополнительное задание, то обязательно отдельно закоммитить код без него.
5. Сдать все через GitHub.

## Пример 1. Сложение N целых чисел

Написать программу, в которой пользователь сначала вводит количество чисел N, а затем сами числа по одному. Программа выводит сумму введенных чисел.

### Указания к выполнению

Компьютер нужен, в первую очередь, для того, чтобы выполнять рутинные, однотипные действия по многу раз.

В программе такое повторение одинаковых действий (инструкций) называется циклом.

**Цикл** – это программная структура, повторяющая заданные инструкции (**тело цикла**), пока выполняется какое-то **условие**.

В большинстве языков программирования используется три типа циклов:

* с **предусловием** (while) – условие проверяется перед телом цикла:

while(<условие>) //пока выполняется условие

{

<тело цикла> //выполняй эти команды

}

* с **постусловием** (do-while) – условие проверяется после тела цикла:

do //выполняй

{

<тело цикла> //эти команды

}

while(<условие>) //пока выполняется условие

* с **параметром** (цикл-счетчик, for) – выполнение цикла регулируется через отдельную переменную-счетчик:

for (<переменная>;<условие>;<приращение>) //пока выполняется условие

{

<тело цикла> //выполняй эти команды

} //изменяй переменную на заданное приращение

Цикл FOR – наиболее простой, чаще всего встречается в таком виде:

for (int i = 0; i < 10; i++) //пока i меньше 10

{

//для примера – выведи i в консоль

Console.Writeln(i);

} //здесь выполняется i++ - увеличить i на 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№ итерации** | **i** | **Условие** | | **i++** |
|  | 0 | 0 < 5 | true | 1 |
|  | 1 | 1 < 5 | true | 2 |
|  | 2 | 2 < 5 | true | 3 |
|  | 3 | 3 < 5 | true | 4 |
|  | 4 | 4 < 5 | true | 5 |
|  | 5 | 5 < 5 | false  конец цикла | - |

Т.е. команда в теле цикла будет повторяться одинаковая, но результат будет разный, т.к. меняется значение i.

Многие задачи можно решить через любой цикл, но в некоторых случаях подходит только while (сложное условие, которое «не помещается» в for) или только do-while (проверку нужно выполнять именно после тела цикла).

Задачу из примера можно реализовать через любой цикл. Мы рассмотрим все три варианта.

Найти сумму некоторых чисел – чрезвычайно простая и популярная задача в программировании, независимо от того, что эти числа означают: баллы за задачи, заработную плату сотрудников, стоимость товаров в чеке и т.д.

Поскольку в условии не указано иное, суммируемые числа будем считать целыми, типа int.

Начнем **с цикла for**.

Алгоритм работы в данном случае удобнее изобразить в виде блок-схемы (более подробную информацию о блок-схемах найдите в сети самостоятельно).

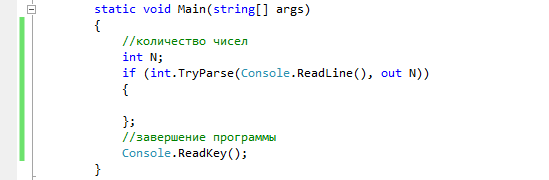
На каждой итерации значение x перезаписывается на следующее и прибавляется к общей сумме.

|  |  |
| --- | --- |
| 1. Ввести количество чисел N. | N  Sum  Sum = 0  Sum += x  x  i=0; i<N; i++ |
| 1. До начала цикла сумма Sum = 0. |
| 1. Пока номер числа i < N повтори: |
| * 1. Ввести следующее значение x. |
| * 1. Прибавить x к сумме Sum. |
| 1. Вывести сумму. |

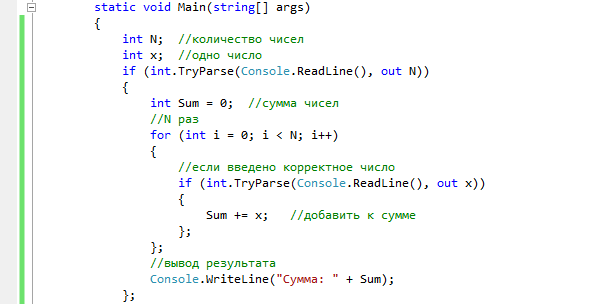
* Как по-другому записать Sum += x? Почему в условии i<N, а не i<=N?

Создайте в VisualStudio новое консольное приложение с названием **SumOfNumbers**.

Добавьте в метод Main переменную n и считывание ее значения из консоли. В конец программы сразу добавьте ReadKey.

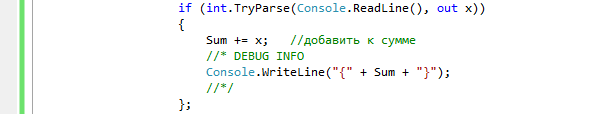


Теперь добавим цикл для ввода значений x и их суммирования.

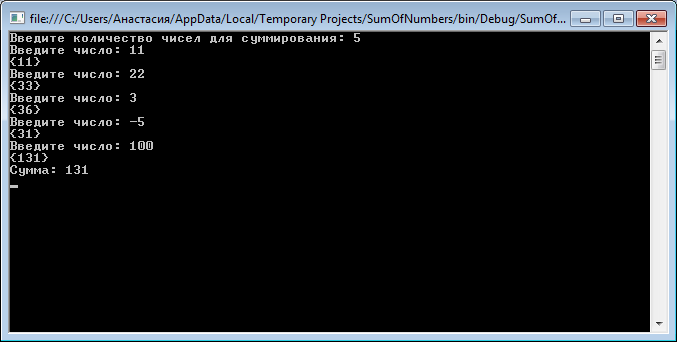


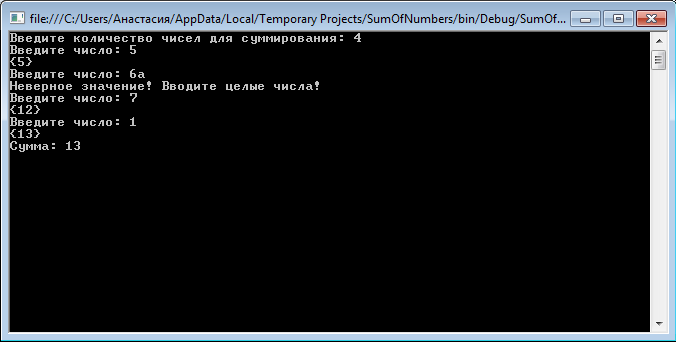
Для того чтобы было проще контролировать правильность программы, добавим вывод промежуточного результата.

Поскольку этот фрагмент не нужен в окончательной версии, обрамим его комментарием.



Самостоятельно добавьте вывод в консоль комментариев для пользователя, поясняющих, что нужно ввести:

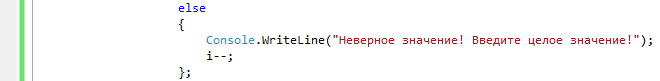




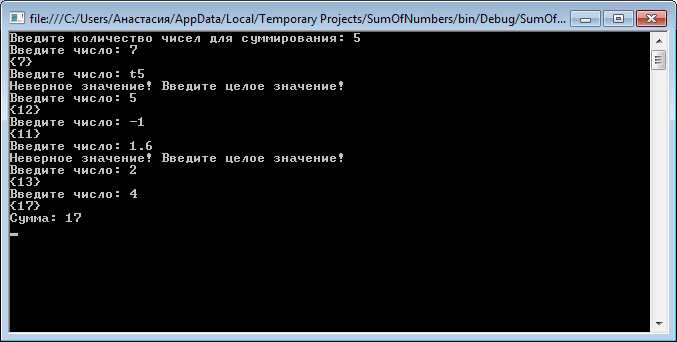
* Что будет, если пользователь введет отрицательное N?

Как видно из примера, неправильно введенное число просто пропускается. Немного изменим программу, чтобы неверное значение было запрошено у пользователя заново.

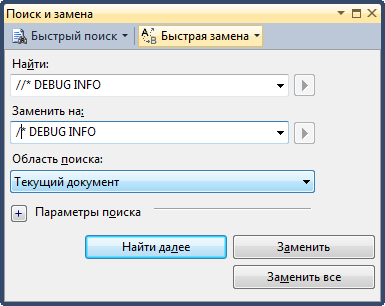
Для этого нам придется уменьшить значение i, как будто мы еще не вводили это число:

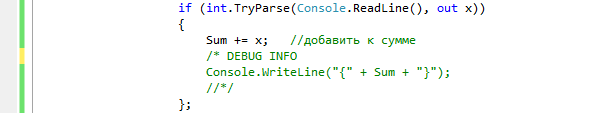


Замечание – изменение значения переменной-счетчика в теле цикла не приводит к ошибкам, но на практике этого стараются избегать. В таком случае лучше использовать цикл while.



После тестирования все фрагменты, отмеченные как DEBUG INFO (если их много), можно закомментировать через замену //\* на /\*.





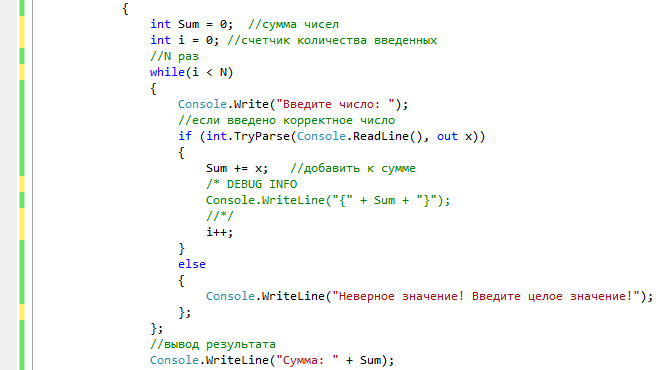
Выполните коммит программы с комментарием «Л3.П1 Суммирование чисел в цикле for».

Изменим программу так, чтобы суммирование выполнялось через цикл while.

Изменения потребуются не очень большие:

1. переменную i объявить до начала цикла;
2. заменить цикл for на while, оставив только условие;
3. выполнить инкремент i++ в теле цикла.

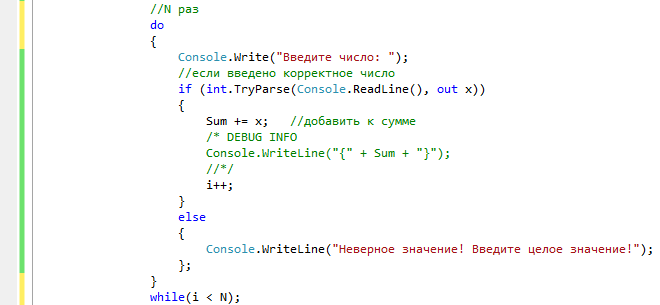
При этом нам больше не потребуется уменьшать значение i--, если было введено неверное значение. Мы просто не будет его увеличивать в этом случае.



* Что будет, если удалить из тела цикла i++ (это называется бесконечный цикл)?

Протестируйте работу программы и выполните коммит программы с комментарием «Л3.П1 Суммирование чисел в цикле while».

Наконец, реализуем программу через цикл с постусловием do-while. Для этого достаточно перенести проверку условия в конец цикла, а в начало добавить ключевое слово do.



* Как будет работать этот цикл, если пользователь введет N равное 0?

Протестируйте работу программы и выполните коммит программы с комментарием «Л3.П1 Суммирование чисел в цикле do-while».

Таким образом, цикл с постусловием выполняется как минимум 1 раз, а циклы с предусловием и с параметром могут не выполниться ни одного раза. В цикле с параметром гораздо менее вероятно получение «бесконечного цикла», т.к. изменение переменной прописывается прямо в заголовке. В while и do-while легко забыть строку, отвечающую за изменение значения, или неправильно записать условие.

### Задание на самостоятельную работу

#### Задание 1

Написать программу, вычисляющую факториал числа (произведение чисел от 1 до этого числа):

n! = 1 × 2 × 3 × ... × n

#### Задание 2

Написать программу, вычисляющую количество и сумму цифр введенного числа.

*Подсказка*:

чтобы узнать последнюю цифру числа, нужно взять остаток от его деления на 10:

x % 10

чтобы отбросить последнюю цифру числа, нужно разделить его на 10 нацело:

x / 10

Пример:

521 % 10 = 1

521 / 10 = 52

52 % 10 = 2

52 / 10 = 5

## Пример 2. Поиск максимального значения в массиве

Написать программу для поиска максимального значения во введенном пользователем массиве. Определить номер максимального элемента.

### Указания к выполнению

**Массив (array)** – это составной тип данных, состоящий из идущих подряд значений одинакового типа. Переменную-массив можно рассматривать как несколько пронумерованных переменных одинакового типа.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Массив* | | | | | | | |
| *Индекс* |  |  |  |  |  |  |  |
| Элемент | 5 | -2 | 1 | 0 | 9 | 6 | 9 |

Объявление массива целых чисел:

int[] arr;

double[] numbers;

string[] names;

Отдельное значение в массиве называется **элементом массива**. Порядковый номер элемента начиная с 0 называется **индексом** и записывается в квадратных скобках:

arr[0]

arr[5]

arr[i]

Все элементы массива сразу можно задать, перечислив в фигурных скобках через запятую:

int[] arr = {5, -2, 1, 6, 9};

Такой способ инициализации применяется редко. Обычно значения вводятся или считываются из файла по одному (в цикле).

Однако сначала массив все равно нужно инициализировать пустыми значениями, задав его длину:

int[] arr = new int[5];

Внимание! Попытка обратиться к неинициализированному массиву, а также неверное значение индекса вызывает исключение и вылет программы.

Длину массива можно узнать через свойство Length:

arr.Length

Ввод и вывод значений массива через цикл for:

int[] arr = new int[5];

for (int i = 0; i < arr.Length; i++)

{

arr[i] = int.Parse(Console.ReadLine());

Console.PrintLine(arr[i]);

}

Суммирование элементов массива:

int[] arr = new int[5];

for (int i = 0; i < 5; i++)

{

arr[i] = int.Parse(Console.ReadLine());

Console.PrintLine(arr[i]);

}

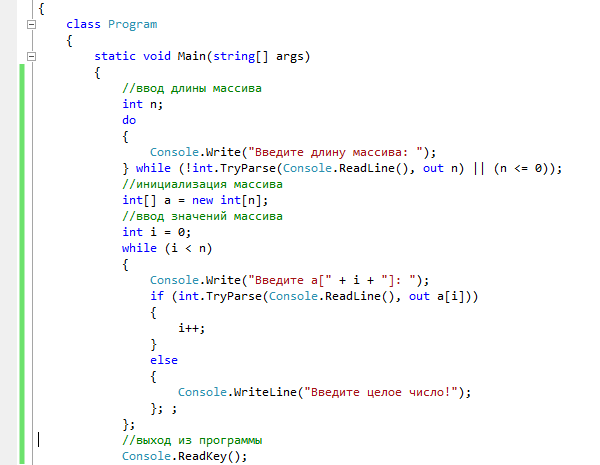
Поиск максимального значения – еще одна классическая задача программирования.

Для этого необходимо перебрать все элементы массива и сравнить их. Вспомните задачу из прошлой лекции, в которой находилось максимальное значение из трех. Одним из способов было сначала сравнить два числа, а потом третье сравнить с большим из них. Для массива используется тот же подход.

|  |  |
| --- | --- |
| 1. Ввести размер массива N и сам массив. | N, a  Max  max = a[0]  max = a[i]  i=1; i<N; i++  a[i]>max  да  нет |
| 1. Проинициализировать максимальное значение из первого элемента массива. |
| 1. Начиная со второго элемента, пока номер числа i < N повтори: |
| * 1. Если текущий элемент a[i] больше максимального, то |
| записать a[i] в максимальное. |
| 1. Вывести максимальное. |

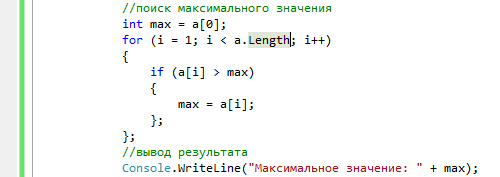
Создайте новое консольное приложение с именем ArrayMax.

Добавьте код для ввода длины массива, его значений и паузы перед выходом.



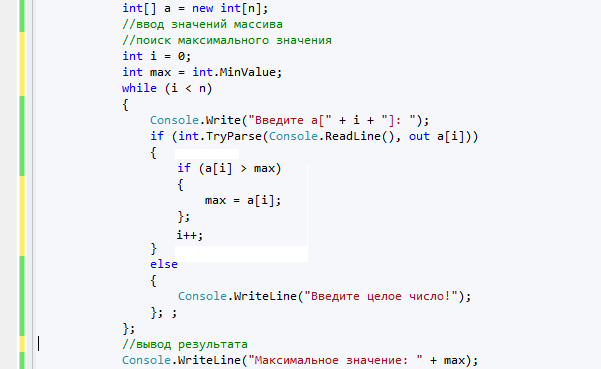
* Почему при контроле ввода длины массива используется цикл do-while? Можно ли поменять условия местами ((n <= 0) || !int.TryParse(Console.ReadLine(), out n))?

Добавьте код для поиска максимального элемента:

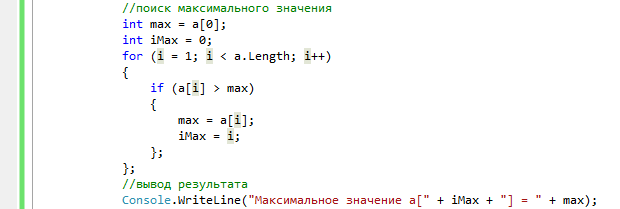


Выполните коммит программы с комментарием «Л2.П2 Поиск максимального значения в массиве».

Примечание – Сейчас в программе присутствует два цикла, while для ввода значений и for для поиска максимума. Их можно объединить в один, сравнивая значение с максимальным сразу после ввода. Это сократит длину программы и немного ускорит ее работу, т.к. мы будем проходить массив не два раза, а один. Но зато снизится читаемость программы и нарушится декомпозиция на две отдельных части: блок ввода данных и блок обработки.



Добавим в программу сохранение индекса максимального элемента в отдельную переменную iMax.



Выполните коммит программы с комментарием «Л2.П2 Поиск индекса максимального значения в массиве».

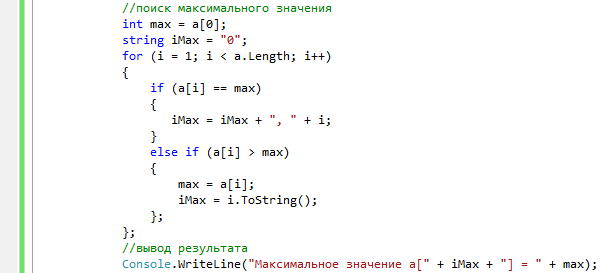
Однако, этот код будет находить только одно максимальное значение. Если в массиве максимальное значение встречается несколько раз (например, {0, 5, **8**, 3, 2, **8**, 4} – индексы {2, 5}), то будет выведен номер только первого из них (в примере 2).

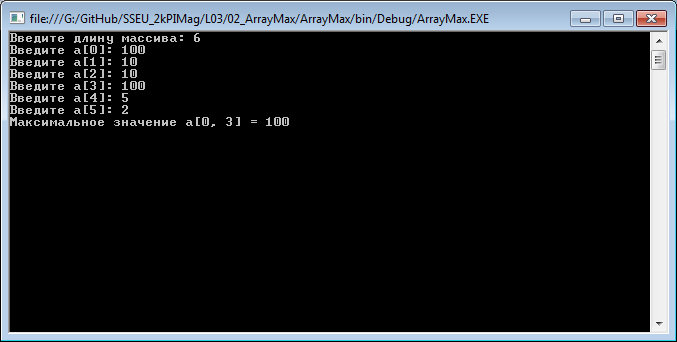
Перечень индексов всех максимальных значений можно сам рассматривать как массив. Но массивы в C# имеют постоянную длину: мы не сможем добавить в него элементы, если вдруг найдем еще одно максимальное значение.

Варианты решения:

1. задать массиву индексов размер «с запасом» (максимальных значений не может быть больше, чем элементов в исходном массиве), но это лишние затраты памяти;
2. использовать другой тип данных – список (List): это хорошее решение, но списки мы рассмотрим позже;
3. сохранять перечень индексов в виде строки текста: этим способом мы и воспользуемся, хотя оно подходит не для всех задач (зависит от того, зачем нужны максимальные значения).

Если найденное значение совпадает с максимальным, то его нужно добавить к списку индексов. А если больше максимального – то начать список заново.





Выполните коммит программы с комментарием «Л2.П2 Поиск индексов всех максимальных значений в массиве».

* Самостоятельно изучите свойства и методы для работы с массивами, которые выводятся в подсказке, если набрать точку после имени массива (там же, где и Length). Как можно найти максимальное значение в одну строчку? В чем преимущества нашего решения?

### Задание на самостоятельную работу

#### Задание 3

Измените программу так, чтобы она находила минимальное значение и заменяла все его вхождения на 0.

## Пример 3. Сортировка массива

Выполнить сортировку элементов массива по возрастанию.

### Указания к выполнению

**Сортировка** массива, т.е. расположение его элементов **по возрастанию** (или убыванию), – еще одна из популярнейших задач программирования.

Существует множество методов сортировки массивов. Одни из них более быстрые, другие более простые.

* Найдите в сети сравнение скорости работы нескольких алгоритмов сортировки массивов.

Мы рассмотрим один из базовых алгоритмов – метод «пузырька», когда бо́льшие значения словно «поднимаются» к концу массива.

Смысл алгоритма заключается в том, что мы последовательно просматриваем элементы массива, и если предыдущий элемент больше текущего, то их нужно поменять местами (чтобы бо́льшее значение оказалось ближе к концу).

После первого прохода массива, самым последним окажется максимальное значение.

Затем проход по массиву повторяется, но уже без последнего (максимального) значения.

N, a

a

buf = a[j];

a[j] = a[j-1];

a[j-1] = buf;

j=1; j<N-i; j++

a[j-1]>a[j]

i=0; i<N-1; i++

да

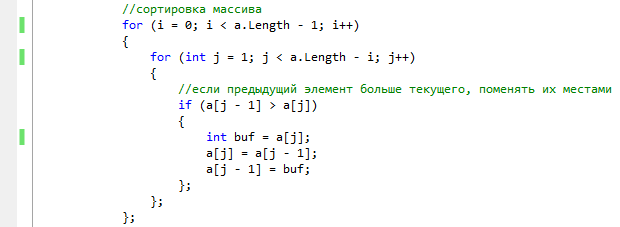
нет

Таким образом, мы получили **вложенный цикл**. Переменная j отвечает за однократный проход по массиву от начала до последнего максимального, а переменная i – за повторение этих проходов.

Создайте новый проект и назовите его SortArray.

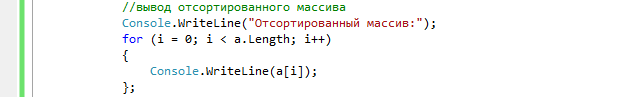
Самостоятельно добавьте ввод значений массива по аналогии с предыдущим примером.

Добавьте алгоритм сортировки:



* Изменится ли результат программы, если > заменить на >=? Какой вариант лучше?

Добавьте вывод отсортированного массива:



Выполните коммит программы с комментарием «Л2.П3 Сортировка массива методом пузырька».

### Задание на самостоятельную работу

#### Задание 5

Напишите программу, выполняющую «шейкерную» сортировку по убыванию.

*Дополнительное задание*: Добавьте возможность выбора направления сортировки (по возрастанию или по убыванию).

## Пример 4. Консольный калькулятор (расширенная версия)

Изменить пример консольного калькулятора из предыдущей лекции так, чтобы он работал не с двумя числами, а с произвольным количеством (как настоящий калькулятор). Вычисления продолжаются до тех пор, пока пользователь не наберет команду выхода (quit или exit).

### Указания к выполнению

Мы расширим функционал калькулятора с помощью циклов. Программа продолжит запрашивать действия и значения до тех пор, пока пользователь не завершит ее работу.

Скопируйте последнюю версию консольного калькулятора из предыдущей лекции в новый проект ConsoleCalc2.

Наша программа становится довольно большой, а редактировать ее как один сплошной текст становится неудобно.

Выделим часть программы в отдельную функцию. **Функция** – это подпрограмма, которая может принимать входные **аргументы** и возвращать результирующее **значение**.

Одна функция в нашей программе уже есть – это функция Main. Она не возвращает значения (тип void).

*Краткий* синтаксис функции:

[static] <возвращаемый тип> <имя функции>(<список аргументов>)

{

<тело функции>

[return <возвращаемое значение>;]

}

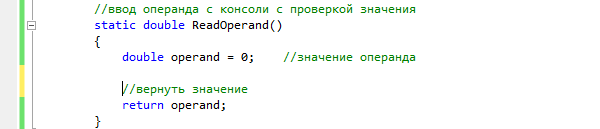
Смысл директивы static будет рассмотрен, когда мы изучим классы. Сейчас оно нам необходимо, т.к. основная функция Main также является статической.

Если функция возвращает значение (результат своей работы), то в ней обязательно должен присутствовать хотя бы один return. В данном случае return завершает работу не всей программы, а только этой функции.

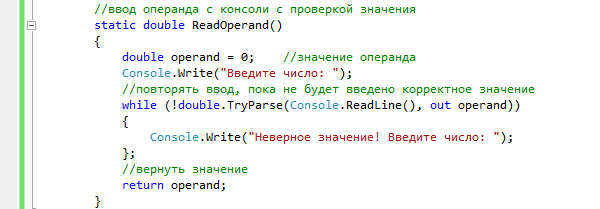
Добавим функцию для ввода и проверки значения операнда. Ввод продолжается, пока пользователь не введет корректное число.

Назовем функцию ReadOperand. У нее нет аргументов (значение вводится с клавиатуры, а не сообщается из программы), возвращаемый тип – double (значение операнда).

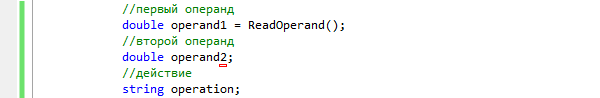
Добавьте описание функции перед Main.



Добавьте цикл с проверкой введенного значения:



Теперь мы можем использовать эту функцию для считывания операндов:



Второй операнд мы будем считывать после операции. Отдельная переменная result нам больше не понадобится: мы будем помещать результат вычислений в operand1, чтобы с ним можно было выполнить еще одно действие.

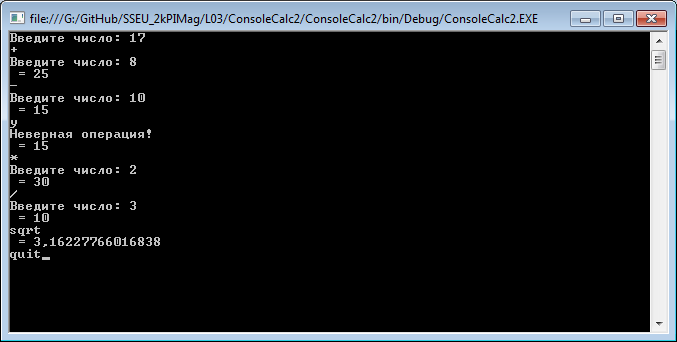
Также нам не потребуются проверки if(...TryParse(...)), т.к. мы вынесли их в функцию.

* + 1. Добавьте цикл do-while для повторения ввода операций.
    2. Замените ввод operand2 через функцию ReadOperand.
    3. Результат вычислений записывайте в operand1.
    4. Выводите на экран только результат из operand1.
    5. Добавьте в switch проверку на введенные команды quit или exit.
    6. Ввод неверной операции больше не должен завершать работу программы.
    7. В конце программы больше не требуется ReadKey, т.к. выход происходит по команде.



* Как сократить запись вычислений для операций с двумя операндами?

Пример работы программы:



Для большей гибкости добавим в нашу программу регистронезависимость, т.е. позволим вводить команды и строчными, и заглавными буквами (quit, Quit, QUIT, qUIT).

Для этого мы не будем менять условия проверки – вариантов слишком много, чтобы перечислять их все. Просто преобразуем введенную операцию в нижний регистр через встроенную функцию ToLower:



Существует и метод перевода в верхний регистр ToUpper.

Выполните коммит программы с комментарием «Л3.П4 Консольный колькулятор».

### Задание на самостоятельную работу

#### Задание 5

Разработать консольное приложение, проверяющее попадание в мишень. Пользователь выполняет N выстрелов по мишени путем ввода пары координат x и y. Радиусы окружностей и очки, начисляемые за попадание в каждую из них, показаны на рисунке.

Программа должна после каждого выстрела выводить очки за этот выстрел и общую сумму набранных очков. В конце игры выводится сводная таблица с координатами и очками по каждому выстрелу, а также координаты самого близкого к центру мишени выстрел.

5

10

15

25

**100**

**50**

**20**

**10**

*Дополнительное задание*: Очки и радиусы окружностей также хранить в массивах, а проверять попадание через циклы. Добавить 5 очков за попадание в окружность радиусом 30.