ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Утверждаю

Заместитель директора по УМР

\_\_\_\_\_\_\_\_\_\_\_\_/Е.А. Родзик

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Методические рекомендации по выполнению   
учебной-практической работы №2

учебной практики   
*УП 05. Программирование*

*Тема: «Функции»*

г. Томск – 2020 г

РАССМОТРЕННО

на заседании ПЦК

«Информационные системы и программирование»

протокол №\_\_\_\_\_\_

от «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_20\_\_\_г.

Председатель ПЦК

\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ Фунтиков М.Н.Рекомендации разработаны на основе Федерального государственного образовательного стандарта (далее – ФГОС) по специальностям среднего профессионального образования (далее СПО) 09.02.07 – «Информационные системы и программирование» и в соответствии с примерной основной образовательной программой.

Организация-разработчик:

ОГБОУ СПО «Томский техникум информационных технологий»

Разработчики:

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Сидиков И.Д.

\_\_\_\_\_\_\_\_\_\_\_\_\_/ Владимировна А. В.

**СОДЕРЖАНИЕ**

[1. Общая характеристика учебной-практической работы №2 4](#_Toc22557890)

[2. Краткие теоретические сведения 5](#_Toc22557891)

[3. Задания для выполнения 22](#_Toc22557892)

[4. Индивидуальные задания 23](#_Toc22557893)

[5. Контрольные вопросы 25](#_Toc22557894)

[6. Рекомендованная литература 26](#_Toc22557895)

[ПРИЛОЖЕНИЕ А 27](#_Toc22557896)

[ПРИЛОЖЕНИЕ Б 28](#_Toc22557897)

# ОБЩАЯ ХАРАКТЕРИСТИКА УЧЕБНОЙ-ПРАКТИЧЕСКОЙ РАБОТЫ №2

**Тема:** «Функции».

**Цель работы:** получение первоначальных навыков по работе с функциями на языке программирования C#

**Проверяемые компетенции:**

ОК 1, ОК 2, ОК 4, ОК 9, ПК11.2.

**Инструкция по выполнению:**

1. Ознакомьтесь с теоретическим материалом, необходимым для выполнения практической работы.
2. Выполните предложенное практическое задание.
3. Оформите отчет по учебной практике (шаблон представлен в приложении)
4. Отправьте отчет на проверку.

**Время выполнения заданий:** 6 часов.

**Критерии оценки:**

|  |  |  |
| --- | --- | --- |
| **№** | **Критерий** | **Баллы** |
| 1 | Отчет соответствует предложенному шаблону | 1 |
| 2 | Задание выполнено правильно | 2 |
| 3 | В коде присутствует функции | 4 |
| 4 | Составлена блок схема к программе | 2 |
| 5 | Программа работает правильно без сбоев | 2 |
| 6 | Студент ответил на контрольные вопросы по лабораторной работе | 2 |
| 7 | Оформление отчета соответствует требованиям (шрифт, поля, отступы, интервалы, оформление рисунков, автоматическое оглавление) | 1 |
| 8 | Своевременность выполнения задания | 1 |
| Итого | | 15 |

**Перевод в пятибалльную систему оценивания:**

«отлично» - 13 - 15 баллов

«хорошо» - 10 -12 баллов

«удовлетворительно» - 7-9

«неудовлетворительно» <7 баллов

# КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## **Введение**

Первыми формами модульности, появившимися в языках программирования, были процедуры и функции. Они позволяли задавать определенную функциональность и многократно выполнять один и тот же параметризованный программный код при различных значениях параметров. Поскольку функции в математике использовались издавна, то появление их в языках программирования было совершенно естественным. Уже с первых шагов процедуры и функции позволяли решать одну из важнейших задач, стоящих перед программистами, - задачу повторного использования программного кода. Встроенные в язык функции давали возможность существенно расширить возможности языка программирования. Важным шагом в автоматизации программирования было появление библиотек процедур и функций, доступных из используемого языка.

Долгое время процедуры и функции играли не только функциональную, но и архитектурную роль. Весьма популярным при построении программных систем был метод функциональной декомпозиции «сверху вниз», и сегодня еще играющий важную роль. Но с появлением объектно-ориентированного программирования (ООП) архитектурная роль функциональных модулей отошла на второй план. Для объектно-ориентированных языков, к которым относится и язык C#, в роли архитектурного модуля выступает класс. Программная система строится из модулей, роль которых играют классы, но каждый из этих модулей имеют содержательную начинку, задавая некоторую абстракцию данных.

Прежнюю роль библиотек процедур и функций теперь играют библиотеки классов. Библиотека классов FCL, доступная в языке C#, существенно расширяет возможности языка. Знание классов этой библиотеки и методов этих классов совершенно необходимо для практического программирования на C# с использованием всей его мощи.

Процедуры и функции связываются с классом, они обеспечивают функциональность данных класса и называются методами класса. Главную роль в программной системе играют данные, а функции лишь служат данным. Напомним еще раз, что в C# процедуры и функции существуют только как методы некоторого класса, они не существуют вне класса. В данном контексте понятие класс распространяется и на все его частные случаи - структуры, интерфейсы, делегаты.

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

**Функция отличается от процедуры двумя особенностями:**

1. Она всегда вычисляет некоторое значение, возвращаемое в качестве результата функции;

2. И вызывается в выражениях.

**Процедура C# имеет свои особенности:**

1. Она возвращает формальный результат void, указывающий на отсутствие результата;

2. Вызов процедуры является оператором языка;

3. Имеет входные и выходные аргументы, причем выходных аргументов - ее результатов - может быть достаточно много.

Обычно метод предпочитают реализовать в виде функции тогда, когда он имеет один выходной аргумент, рассматриваемый как результат вычисления значения функции. Возможность вызова функций в выражениях также влияет на выбор в пользу реализации метода в виде функции. В других случаях метод реализуют в виде процедуры.

## **Функции**

Функции позволяют вам выделить часть кода программы и вызвать его из других частей программы. Вам может быть трудно повторять части кода во многих местах, поэтому в таких случаях на помощь приходят функции. В C# их обычно декларируют так:

|  |
| --- |
| С# |
| <visibility> <return type> <name>(<parameters>)  {  <function code>  } |

Чтобы вызвать функцию, вы просто пишете ее имя, открывающуюся скобку, параметры, если они есть, и закрывающуюся скобку. Вот так:

|  |
| --- |
| С# |
| DoStuff(); |

Это пример нашей функции DoStuff():

|  |
| --- |
| С# |
| public void DoStuff()  {  Console.WriteLine("I'm doing something...");  } |

Первая часть public обозначает область видимости переменной, и она является выборным параметром. Если вы его не определяете, то функция будет private. Подробнее об этом позже. Следующий параметр - возвращаемый тип. Если возможен допустимый в C# тип или, как показано здесь, void. Это означает, что функция не возвращает ничего. Также эта функция не имеет параметров, что видно по пустым скобкам за названием функции. Чтобы оживить картину, внесем изменения:

|  |
| --- |
| С# |
| public int AddNumbers(int number1, int number2)  {  int result = number1 + number2;  return result;  } |

Мы изменили почти все. Функция теперь возвращает целое значение, она принимает два целых параметра и вместо того, чтобы что-то выводить, меняет вычисление и тогда возвращает результат. Это значит, что мы можем добавить два числа из разных мест нашего кода, просто вызвав эту функцию вместо того, чтобы каждый раз описывать вычисление. Этот маленький пример показывает, что функции помогают сэкономить множество времени и усилий:

|  |
| --- |
| С# |
| int result = AddNumbers(10, 5);  Console.WriteLine(result); |

Как было сказано, эта функция возвращает значение, которое должна была вычислить. Если декларируется тип возвращаемого значения функции не void, мы сами должны приложить усилия, описав, что именно должна вернуть функция. Вы можете удалить строку с возвращаемым значением из кода и уведите, что сделает программа:

'AddNumbers(int, int)': не все части кода возвращают значение.

Компилятор напоминает нам, что у нас есть функция, которая ничего не возвращает, хотя мы обещали. И компилятор достаточно умный! Вместо удаления строки, попробуйте что-то вроде этого:

|  |
| --- |
| С# |
| public int AddNumbers(int number1, int number2)  {  int result = number1 + number2;  if(result > 10)  {  return result;  }  } |

Вы увидите точно такую же ошибку, но почему? Потому что нет гарантии, что наш if-оператор получит значение "true" и строка "return" будет выполнена. Вы можете это исправить, введя в конце кода вторую строку "return".

|  |
| --- |
| С# |
| public int AddNumbers(int number1, int number2)  {  int result = number1 + number2;  if(result > 10)  {  return result;  }  return 0;  } |

Это решит созданную нами проблему и продемонстрирует, что в функции может быть не один оператор return. Как только он будет достигнут, транслятор кода покинет функцию. Это означает, что как только результат превысит 10, return 0", никогда не будет достигнут.

## **Локальные функции**

Начиная с версии 7.0 в языке C# поддерживаются локальные функции. Локальные функции представляют собой частные методы типа, вложенные в другой элемент. Они могут вызываться только из того элемента, в который вложены. Ниже перечислены элементы, в которых можно объявлять и из которых можно вызывать локальные функции:

* Методы, в частности методы итератора и асинхронные методы
* Конструкторы
* Методы доступа свойств
* Методы доступа событий
* Анонимные методы
* Лямбда-выражения
* Методы завершения
* Другие локальные функции

Тем не менее локальные функции нельзя объявлять внутри элемента, воплощающего выражение.

|  |
| --- |
| Примечание!  В некоторых случаях для реализации возможностей, поддерживаемых локальными функциями, также можно использовать лямбда-выражения. |

Применение локальных функций позволяет сделать предназначение кода более понятным. Другие пользователи, читающие ваш код, смогут видеть, что соответствующий метод вызывается только внутри того метода, в который он вложен. В случае с командными проектами это также гарантирует, что другой разработчик не сможет ошибочно вызвать метод напрямую из любого другого места в классе или структуре.

## **Синтаксис локальной функции**

Локальная функция определяется как вложенный метод внутри содержащего ее элемента. Ниже приведен синтаксис определения локальной функции:

|  |
| --- |
| С# |
| <modifiers: async | unsafe> <return-type> <method-name> <parameter-list> |

Локальные функции могут использовать модификаторы async и unsafe.

Обратите внимание, что все локальные переменные, определенные в содержащем функцию элементе (включая параметры метода), доступны в локальной функции.

В отличие от определения метода, определение локальной функции не может содержать модификатор доступа к элементу. Поскольку все локальные функции являются частными, при использовании модификатора доступа (например, ключевого слова private) возникает ошибка компилятора CS0106, "Модификатор "private" недопустим для этого элемента".

|  |
| --- |
| Примечание!  В версиях C# ниже 8.0 локальные функции не могут содержать модификатор static. При использовании ключевого слова static возникает ошибка компилятора CS0106, "Модификатор "static" недопустим для этого элемента". |

Кроме того, к локальной функции, а также ее параметрам и параметрам типа, нельзя применять атрибуты.

В следующем примере определяется локальная функция AppendPathSeparator, которая является частной для метода GetText:

|  |
| --- |
| С# |
| using System;  using System.IO;  class Example  {  static void Main()  {  string contents = GetText(@"C:\temp", "example.txt");  Console.WriteLine("Contents of the file:\n" + contents);  }    private static string GetText(string path, string filename)  {  var sr = File.OpenText(AppendPathSeparator(path) + filename);  var text = sr.ReadToEnd();  return text;    // Declare a local function.  string AppendPathSeparator(string filepath)  {  if (! filepath.EndsWith(@"\"))  filepath += @"\";  return filepath;  }  }  } |

## **Локальные функции и исключения**

Полезной особенностью локальных функций является то, что они допускают немедленную обработку исключений. В случае с итераторами метода исключения обрабатываются только после перечисления возвращаемой последовательности, а не в момент извлечения итератора. В случае с асинхронными методами любые исключения, возникшие в таком методе, наблюдаются в тот момент, когда возвращаемая задача находится в состоянии ожидания.

В следующем примере определяется метод OddSequence, который перечисляет нечетные числа в заданном диапазоне. Поскольку он передает в метод перечислителя OddSequence число больше 100, этот метод вызывает исключение ArgumentOutOfRangeException. Как видно из выходных данных этого примера, исключение обрабатывается только в момент перебора чисел, а не при извлечении перечислителя.

|  |
| --- |
| С# |
| using System;  using System.Collections.Generic;  class Example  {  static void Main()  {  IEnumerable<int> ienum = OddSequence(50, 110);  Console.WriteLine("Retrieved enumerator...");    foreach (var i in ienum) //Line 11  {  Console.Write($"{i} ");  }  }  public static IEnumerable<int> OddSequence(int start, int end)  {  if (start < 0 || start > 99)  throw new ArgumentOutOfRangeException("start must be between 0 and 99.");  if (end > 100)  throw new ArgumentOutOfRangeException("end must be less than or equal to 100."); //Line 22  if (start >= end)  throw new ArgumentException("start must be less than end.");    for (int i = start; i <= end; i++)  {  if (i % 2 == 1)  yield return i;  }  }  }  // The example displays the following output:  // Retrieved enumerator...  //  // Unhandled Exception: System.ArgumentOutOfRangeException: Specified argument was out of the range of valid values.  // Parameter name: end must be less than or equal to 100.  // at Sequence.<GetNumericRange>d\_\_1.MoveNext() in Program.cs:line 11  // at Example.Main() in Program.cs:line 22 |

Вместо этого исключение может быть вызвано при выполнении проверки до того, как будет извлечен итератор, путем возврата итератора из локальной функции, как показано в следующем примере.

|  |
| --- |
| С# |
| using System;  using System.Collections.Generic;  class Example  {  static void Main()  {  IEnumerable<int> ienum = OddSequence(50, 110); //Line 8  Console.WriteLine("Retrieved enumerator...");    foreach (var i in ienum)  {  Console.Write($"{i} ");  }  }  public static IEnumerable<int> OddSequence(int start, int end)  {  if (start < 0 || start > 99)  throw new ArgumentOutOfRangeException("start must be between 0 and 99.");  if (end > 100)  throw new ArgumentOutOfRangeException("end must be less than or equal to 100."); //Line 22  if (start >= end)  throw new ArgumentException("start must be less than end.");    return GetOddSequenceEnumerator();    IEnumerable<int> GetOddSequenceEnumerator()  {  for (int i = start; i <= end; i++)  {  if (i % 2 == 1)  yield return i;  }  }  }  }  // The example displays the following output:  // Unhandled Exception: System.ArgumentOutOfRangeException: Specified argument was out of the range of valid values.  // Parameter name: end must be less than or equal to 100.  // at Sequence.<GetNumericRange>d\_\_1.MoveNext() in Program.cs:line 8  // at Example.Main() in Program.cs:line 22 |

Локальные функции можно использовать аналогичным образом для обработки исключений вне асинхронной операции. Как правило, при возникновении исключения в асинхронном методе требуется проверить внутренние исключения в AggregateException. Локальная функция реализует моментальный сбой кода, синхронно обеспечивая вызов исключения и наблюдение за ним.

В следующем примере асинхронный метод GetMultipleAsync выполняет приостановку на указанное число секунд, возвращая значение, представляющее собой произведение случайного множителя на это число секунд. Максимальная задержка составляет 5 с. Результат ArgumentOutOfRangeException возвращается в том случае, если значение больше 5. Как видно из следующего примера, исключение, которое возникает при передаче в метод GetMultipleAsync значения 6, инкапсулируется в AggregateException после того, как начинается выполнение метода GetMultipleAsync.

|  |
| --- |
| С# |
| using System;  using System.Threading.Tasks;  class Example  {  static void Main()  {  int result = GetMultipleAsync(6).Result; //Line 8  Console.WriteLine($"The returned value is {result:N0}");  }  static async Task<int> GetMultipleAsync(int secondsDelay)  {  Console.WriteLine("Executing GetMultipleAsync...");  if (secondsDelay < 0 || secondsDelay > 5)  throw new ArgumentOutOfRangeException("secondsDelay cannot exceed 5."); // Line 16    await Task.Delay(secondsDelay \* 1000);  return secondsDelay \* new Random().Next(2,10);  }  }  // The example displays the following output:  // Executing GetMultipleAsync...  //  // Unhandled Exception: System.AggregateException:  // One or more errors occurred. (Specified argument was out of the range of valid values.  // Parameter name: secondsDelay cannot exceed 5.) --->  // System.ArgumentOutOfRangeException: Specified argument was out of the range of valid values.  // Parameter name: secondsDelay cannot exceed 5.  // at Example.<GetMultiple>d\_\_1.MoveNext() in Program.cs:line 16  // --- End of inner exception stack trace ---  // at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)  // at System.Threading.Tasks.Task`1.GetResultCore(Boolean waitCompletionNotification)  // at Example.Main() in C:\Users\ronpet\Documents\Visual Studio 2017\Projects\local-functions\async1\Program.cs:line 8 |

Как и в случае с итератором метода, можно выполнить рефакторинг кода из этого примера таким образом, чтобы реализовать проверку перед вызовом асинхронного метода. Как видно из результатов следующего примера, ArgumentOutOfRangeException не инкапсулируется в AggregateException.

|  |
| --- |
| С# |
| using System;  using System.Threading.Tasks;  class Example  {  static void Main()  {  int result = GetMultiple(6).Result; // Line 8  Console.WriteLine($"The returned value is {result:N0}");  }  static Task<int> GetMultiple(int secondsDelay)  {  if (secondsDelay < 0 || secondsDelay > 5)  throw new ArgumentOutOfRangeException("secondsDelay cannot exceed 5."); // Line 15    return GetValueAsync();    async Task<int> GetValueAsync()  {  Console.WriteLine("Executing GetValueAsync...");  await Task.Delay(secondsDelay \* 1000);  return secondsDelay \* new Random().Next(2,10);  }  }  }  // The example displays the following output:  // Unhandled Exception: System.ArgumentOutOfRangeException:  // Specified argument was out of the range of valid values.  // Parameter name: secondsDelay cannot exceed 5.  // at Example.GetMultiple(Int32 secondsDelay) in Program.cs:line 15  // at Example.Main() in Program.cs:line 8 |

## **Примеры локальных функций**

Локальные функции представляют функции, определенные внутри других методов. Определим и используем локальную функцию:

|  |
| --- |
| С# |
| class Program  {      static void Main(string[] args)      {          var result = GetResult(new int[] { -3, -2, -1, 0, 1, 2, 3 });          Console.WriteLine(result);  // 6          Console.Read();      }        static int GetResult(int[] numbers)      {          int limit = 0;          // локальная функция          bool IsMoreThan(int number)          {              return number > limit;          }            int result = 0;          for(int i=0; i < numbers.Length; i++)          {              if (IsMoreThan(numbers[i]))              {                  result += numbers[i];              }          }            return result;      }  } |

Здесь в методе GetResult определена локальная функция IsMoreThan(), которая может быть вызвана только внутри этого метода. Локальная функция задает еще одну область видимости, где мы можем определять переменные и выполнять над ними действия. В то же время ей доступны все переменные, которые определены в том же методе.

При использовании локальных функций следует помнить, что они не могут иметь модификаторов доступа (public, private, protected). Нельзя определить в одном методе несколько локальных функций с одним и тем же именем, даже если у них будет разный список параметров. Кроме того, не имеет значения, определены локальные функции до своего вызова или после.

Начиная с версии C# 8.0 можно определять статические локальные функции. Их особенностью является то, что они не могут обращаться к переменным окружения, то есть метода, в котором статическая функция определена. Например, перепишем метод GetResult, сделав функцию IsMoreThan статической:

|  |
| --- |
| С# |
| static int GetResult(int[] numbers)  {      // статическая локальная функция      static bool IsMoreThan(int number)      {          int limit = 0;          return number > limit;      }        int result = 0;      for (int i = 0; i < numbers.Length; i++)      {          if (IsMoreThan(numbers[i]))              result += numbers[i];      }      return result;  } |

## **Параметры функций**

Первое, на что мы обратим внимание, это модификаторы out и ref. Программные языки такие, как как C#, различают два параметра вывода результата функции: "по значению" и "по ссылке". В C# текущим является "по значению", что обычно значит, что, когда вы передаете переменную в функцию при ее вызове, то фактически передаете туда копию объекта, а не ссылку на него. Это также означает, что вы можете изменить параметр из функции, не изменяя оригинала объекта, который передаете как параметр.

## **Модификатор ref**

|  |
| --- |
| С# |
| static void Main(string[] args)  {  int number = 20;  AddFive(number);  Console.WriteLine(number);  Console.ReadKey();  }  static void AddFive(int number)  {  number = number + 5;  } |

Мы создаем переменную целого типа integer, присваиваем ей число 20 и тогда применяем метод AddFive(), который должен прибавить к integer 5. Но прибавит ли? Нет. Значение, которое мы присвоили integer внутри функции, никогда не будет вынесено из функции, потому что в функцию мы передали копию значения переменной вместо того, чтобы передать туда ссылку на него. Это простое объяснение работы компилятора C#, и чаще всего такой результат предпочтителен. Однако, в этом случае мы бы хотели изменить значение переменной внутри нашей функции. Поэтому следует использовать ключевое слово ref:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  int number = 20;  AddFive(ref number);  Console.WriteLine(number);  Console.ReadKey();  }  static void AddFive(ref int number)  {  number = number + 5;  } |

Как видите, все, что мы сделали, это добавили ref к определению функции в ее вызове. Если вы сейчас запустите программу, то увидите, что значение number изменилось, поскольку мы вернули его функции при ее вызове.

## **Модификатор out**

Модификатор out работает во многом подобно модификатору ref. Они оба обеспечивают передачу параметра по ссылке, а не по значению, но они характерны двумя важными различиями: значение, передаваемое модификатору ref, должно быть инициализировано до вызова метода - это не относится к модификатору out, который вы можете использовать с неинициализированными значениями. С другой стороны, вы не можете выйти из функции, применив параметр out, не присвоив ему значение. Поскольку вы можете как параметр передать значение без его инициализации, то не можете использовать параметр out внутри функции - вы сможете только присвоить ему новое значение.

Применять параметр out или ref, зависит от ситуации, что вы поймете, когда начнете их использовать. Они оба обычно используются в работе, возвращая то или иное значение из функции C#.

Применение модификатора out похоже на использование модификатора ref, описанное выше. Просто поменяйте ref на out. В предыдущем примере не забудьте удалить присваивание значения переменной number в методе и вместо этого декларируйте его в вызове функции.

## **Модификатор params**

Все наши предыдущие функции принимали фиксированное количество параметров. Однако во многих случаях вам может понадобиться функция, которая принимает произвольное число параметров. Это можно сделать, используя в качестве параметров массив или список, как здесь:

|  |
| --- |
| С# |
| static void GreetPersons(string[] names) { } |

Однако его вызов может быть грубоватым. Более короткая форма выглядит так:

|  |
| --- |
| С# |
| GreetPersons(new string[] { "John", "Jane", "Tarzan" }); |

Она приемлема, но может быть разумнее с применением модификатора params:

|  |
| --- |
| С# |
| static void GreetPersons(params string[] names) { } |

Ее вызов выглядел бы так:

|  |
| --- |
| С# |
| GreetPersons("John", "Jane", "Tarzan"); |

Другая выгода при использовании подхода params в том, что вы так же можете передать ему нулевой параметр. Функции с params могут также принимать параметры long и последним ключевым словом params. Кроме того, в функции может использоваться только один параметр, использующий ключевое слово params. Вот последний и более полный пример:

|  |
| --- |
| С# |
| static void Main(string[] args)  {  GreetPersons(0);  GreetPersons(25, "John", "Jane", "Tarzan");  Console.ReadKey();  }  static void GreetPersons(int someUnusedParameter, params string[] names)  {  foreach(string name in names)  Console.WriteLine("Hello, " + name);  } |

## **Модификатор доступа**

Между модификатором и типом может стоять ключевое слово static, что означает, что функция будет статичной. Подробно говорить о статичных функциях и переменных мы будем в отдельном уроке. Скажу только, что из статичной функции можно вызывать другие функции, если они тоже статичные. Главная функция main – всегда static, поэтому все функции в этом уроке мы будем объявлять также статичными.

Функция может возвращать значение или не возвращать. Если функция, например, возвращает целое число, то нужно указать тип int. Если функция не возвращает никакого значения, то для этого используется ключевое слово void. Функции, которые не возвращают значение, еще называют процедурами.

Называть функции стоит так, чтобы имя отображало суть функции. Используйте глаголы или словосочетания с глаголами. Примеры: GetAge(), Sort(), SetVisibility().

**Аргументы** – это те данные, которые необходимы для выполнения функции. Аргументы записываются в формате [тип] [идентификатор]. Если аргументов несколько, они отделяются запятой. Аргументы могут отсутствовать.

Первая строка функции, где указываются тип, имя, аргументы и т.д. называется заголовком функции.

**Пример функции, которая не возвращает значение:**

Напишем простую функцию, которая будет заменять в массиве строк указанное имя на другое. Данная функция будет принимать три аргумента: массив строк, имя, которое необходимо заменить и новое имя. Так как функция не будет возвращать значение, указываем тип void перед именем функции.

|  |
| --- |
| С# |
| public static void ReplaceName(string[] names, string name, string newName)  {  for (int i=0; i < names.Length; i++)  {  if (names[i] == name)  names[i] = newName;  }  } |

Сама функция очень простая. Проходим в цикле по элементам и смотрим, равен ли элемент указанному имени. Если да, то заменяем его на новое имя.

Функция написана, и теперь используем ее:

|  |
| --- |
| С# |
| class Program  {  public static void ReplaceName(string[] names, string name, string newName)  {  for (int i=0; i < names.Length; i++)  {  if (names[i] == name)  names[i] = newName;  }  }  static void Main(string[] args)  {  string[] names = { "Sergey", "Maxim", "Andrey", "Oleg", "Andrey", "Ivan", "Sergey" };  ReplaceName(names, "Andrey", "Nikolay"); // вызов функции. Все строки "Andrey" в массиве будут заменены на "Nikolay"  ReplaceName(names, "Ivan", "Vladimir");  }  } |

После вызова функции два раза в этой программе, массив будет выглядеть так: "Sergey", "Maxim", " Nikolay ", "Oleg", " Nikolay ", " Vladimir ", "Sergey

**Пример функции, которая возвращает значения:**

Напишем функцию, которая будет находить максимальное число в массиве. Аргумент у этой функции будет один – массив целых чисел. Тип возвращаемого значения – целое число int.

|  |
| --- |
| С# |
| public static int GetMax(int[] array)  {  int max = array[0];  for (int i = 1; i < array.Length; i++)  {  if (array[i] > max)  max = array[i];  }  return max;  } |

Логика функции проста. Создаем переменную max, в которую записываем первый элемент массива. Дальше в цикле сравниваем каждый элемент со значением в max, если элемент больше, чем max, то записываем в max этот элемент. В конце функции используем оператор return, чтобы вернуть наш результат.

Оператор return должен быть обязательно в функции, которая возвращает значение.

Используем нашу функцию:

|  |
| --- |
| С# |
| class Program  {  public static int GetMax(int[] array)  {  int max = array[0];  for (int i = 1; i < array.Length; i++)  {  if (array[i] > max)  max = array[i];  }  return max;  }  static void Main(string[] args)  {  int[] numbers = { 3, 32, 16, 27, 55, 43, 2, 34 };  int max;  max = GetMax(numbers); //вызов такой функции можно использовать при присваивании значения  Console.WriteLine(GetMax(numbers)); // вызов функции также можно использовать как аргумент при вызове другой функции. WriteLine() – тоже функция.  Console.ReadKey();  }  } |

## **Оператор return**

Когда встречается этот оператор, происходит выход из функции и код ниже (если он есть) выполняться не будет (например, в функцию передан такой аргумент, при котором нет смысла выполнять функцию). Он похож на оператор break, который используется для выхода из циклов. Этот оператор также можно использовать и в функциях, которые не возвращают значение. Оператор return допустимо использовать несколько раз в функции, но этого делать не рекомендуется.

# ЗАДАНИЯ ДЛЯ ВЫПОЛНЕНИЯ

1. Решите задание используя функции, по вариантам
2. Спроектируйте блок схему к программе
3. Отобразить в отчете проделанные шаги
4. Ответить на контрольные вопросы
5. Написать вывод о проделанной работе

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

|  |
| --- |
| Вариант 1 – Напишите функцию, которая будет менять в массиве целых чисел все элементы, которые равны указанному значению (аргумент) на противоположное значение по знаку. Например, все элементы массива, которые равны 5, будут меняться на -5. |
| Вариант 2 – Напишите функцию, которая будет находить минимальное число из трех. |
| Вариант 3 – Напишите функцию, которая будет возвращать указанный элемент ряда Фибоначчи. Ряд Фибоначчи – это ряд, в котором каждый следующий элемент равен сумме двух предыдущих. 1 1 2 3 5 8 13 21… Функция принимает порядковый номер элемента, и возвращает соответствующий элемент. |
| Вариант 4 - Напишите программу для поиска индекса максимального элемента массива с использованием рекурсии. |
| Вариант 5 – Написать функцию, возвращающую номер самого последнего элемента из массива, который совпадает с заданным с клавиатуры числом. |
| Вариант 6 - Палиндром — это строка, которая читается одинаково в обоих направлениях. Например, "**tot**" и "**otto**" — короткие палиндромы. Напишите программу, которая запрашивает у пользователя строку и передает ссылку на нее функции Check типа bool. Функция должна возвращать true, если строка является палиндромом, и false — в противном случае. |
| Вариант 7 - Программа многократно запрашивает у пользователя пару чисел до тех пор, пока хотя бы одно из этой пары не будет равно 0. С каждой парой программа должна использовать функцию для вычисления среднего гармонического этих чисел. Функция должна возвращать ответ в main () для отображения результата. Среднее гармоническое чисел — это инверсия среднего значения их инверсий, она вычисляется следующим образом: среднее гармоническое = 2.0 \*х \* у / (х + у). |
| Вариант 8 - Программа выдает запрос на ввод значений часов и минут. Функция main () должна передать эти два значения функции, имеющей тип void, которая отобразит эти два значения в следующем виде: часы : минуты. |
| Вариант 9 - Объявить два целочисленных массива с разными размерами и написать функцию, которая заполняет их элементы значениями и показывает на экран. Функция должна принимать два параметра – массив и его размер. |
| Вариант 10 - Необходимо создать двумерный массив 5 х 5. Далее написать функцию, которая заполнит его случайными числами от 30 до 60. Создать еще две функции, которые находят максимальный и минимальный элементы этого двумерного массива. |

# КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Как работает оператор params?
2. Локальные функции какие у них есть преимущества?
3. Какие бывают модификаторы?

# РЕКОМЕНДОВАННАЯ ЛИТЕРАТУРА

1. Официальная документация Microsoft C# — Текст: электронный // Microsoft [сайт]. — URL: https://docs.microsoft.com/ (дата обращения: 12.03.2020).
2. Сообщество IT-специалистов — Текст: электронный // Habr [сайт]. — URL: https://habr.com/ (дата обращения: 12.03.2020).
3. Сайт о программирование / — Текст: электронный // Metanit [сайт]. — https://metanit.com/ (дата обращения: 12.03.2020).
4. С# Tutotial — Текст: электронный // Tutotial [сайт]. — URL: https://csharp.net-tutorials.com/ (дата обращения: 13.03.2020).

# ПРИЛОЖЕНИЕ А

**Требования к отчету**

Общие требования:

1. Шрифт – Times New Roman, 14 пт.
2. Интервалы: междустрочный – 1,5 строки, интервал до и после абзаца – 0 пт.
3. Отступ первой строки – 1,25
4. Рисунки и подписи к ним выравниваются по центру.

Требования к структуре отчета:

1. Титульный лист
2. Содержание
3. Основная часть:
   1. Задачи
   2. Код
   3. Блок-схемы
   4. Результат работы
4. Выводы по работе
5. Ответы на контрольные вопросы

# ПРИЛОЖЕНИЕ Б

ОГБПОУ «ТОМСКИЙ ТЕХНИКУМ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ»

Отчет по учебной-практической работе №2

учебной практике   
*УП 05. Программирование*

*Тема: «Функции»*

Выполнил:

студент \_\_\_\_\_\_\_ группы

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Проверил:  
преподаватель

Владимирова А.В.

г. Томск – 2020 г