МИНОБРНАУКИ РОССИИ
Федеральное государственное автономное образовательное учреждение высшего образования "Национальный исследовательский Томский государственный университет"

ФАКУЛЬТЕТ ИНФОРМАТИКИ

Поддубная Тамара Николаевна

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
для лабораторного практикума по основам алгоритмизации на языке C#

Томск
2015

Методическое пособие рассмотрено и одобрено методической комиссией факультета информатики.

Декан факультета информатики С.П. Сущенко

Председатель методической комиссии А.В. Приступа

Пособие предназначено для студентов первого курса факультета информатики Томского государственного университета, изучающих дисциплину «Основы алгоритмизации», и соответствует курсу лекций по этой дисциплине. Пособие содержит методические указания для выполнения 10 лабораторных работ на языке C# в среде Visual Studio.NET. Каждое методическое указание включает в себя элементы теоретического материала, необходимого для написания алгоритма, описание возможностей языка C# и снабжено примерами фрагментов программного кода. Это делает пособие доступным и для тех студентов, которые не изучали никакого языка программирования. Пособие может быть использовано студентами также для самостоятельного изучения программирования на языке C#.

СОДЕРЖАНИЕ

1. Создание Windows-приложений в среде VS.NET

Идея использовать графический пользовательский интерфейс (GUI – graphical user interface) и мышь для работы с компьютером появилась задолго до оболочки Windows. Понятие графического интерфейса было первоначально введено еще в начале 1970-х годов инженерами из исследовательского центра корпорации Xerox в г. Пало Альто (PARC – Palo Alto Research Center) в Калифорнии. Один из первых компьютеров, в которых использовались мышь и GUI, назывался Alto. К сожалению, он был очень дорогим, поэтому графический интерфейс пользователя стал популярным только после того, как фирма Apple Computer выпустила компьютер Macintosh в 1984 году. Позже фирма Microsoft разработала операционную систему Windows, построенную на идеях, которые стали популярными благодаря разработкам Apple.

Графическая оболочка Windows делает доступными для использования в программах графические элементы, такие как меню, кнопки, поля для ввода текста и переключатели, позволяя создать легкий в использовании визуальный интерфейс.

Известно, что Windows-приложения легки в освоении и использовании, поэтому все приложения для лабораторных работ по курсу «Основы алгоритмизации» мы рекомендуем выполнять как Windows-приложения. В качестве языка программирования рекомендуем выбрать язык C# (произносится как «си-шарп»), являющийся родным языком среды Visual Studio.NET (VS.NET). Visual Studio.NET – это интегрированная среда для быстрой разработки приложений (Rapid Application Development, RAD) фирмы Microsoft.

Про среду VS.NET и платформу .NET (произносится «дот-нет») изучить самостоятельно по литературе

Все приложения, разрабатываемые в VS.NET, организуются в **проекты**. Рекомендуемый порядок работы описан далее на примере создания первого проекта, предназначенного просто для знакомства со средой VS.NET, некоторыми элементами программирования на языке C# и способами написания программного кода.

По традиции, идущей от авторов книги «Язык программирования C» Брайана Кернигана и Дениса Ритчи, в нашем первом проекте мы создадим Windows-окно, в которое выведем текст «Hello world!», текущую дату и время, и кое-что еще по желанию.

1. Запустите VS.NET;
2. Создайте новое приложение, выбрав команду меню File-New-Project и в открывшемся окне – Windows Applications для языка C#;
3. В этом же окне в соответствующих строках (в нижней части окна) укажите имя для проекта в поле Name и имя папки, которую VS.NET создаст для сохранения всех файлов проекта, в поле Location. Рекомендуется заменить имена, предлагаемые VS.NET, на другие, которые соответствуют смыслу проекта. Например, в нашем случае для

проекта, который предназначен для знакомства (acquaintance), имя может быть такими – WinApplAcquaintance, а для папки – ApplAcquaintance.

4. Как только VS.NET выполнит все необходимые действия по сохранению проекта, на экране отобразятся все окна, которые будут использоваться на этапе дизайна.

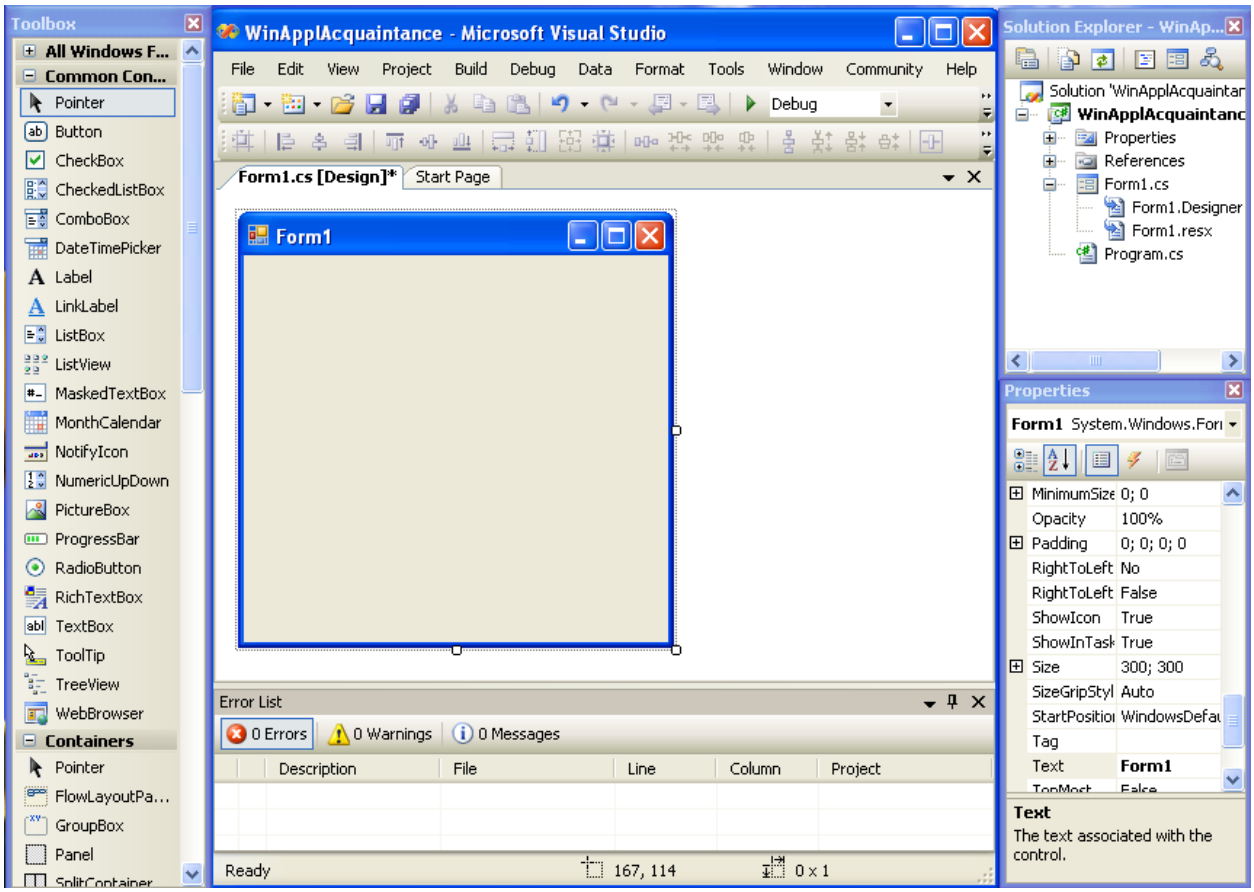


Рис. 1. Пустая форма

Пустая форма будет использоваться далее для размещения на ней стандартных управляющих элементов Windows, таких как статический текст (Label), поля ввода текста (TextBox), кнопки (Button) и др. Эти элементы размещены в окне-панели инструментов ToolBox (слева от пустой формы). Щелкнув по управляющему элементу в панели инструментов, можно затем «перетащить» его на форму, поместить в желаемое место и задать желаемые свойства. Для этого используется окно свойств (Properties), расположенное справа от пустой формы. Вверху над окном свойств располагается окно с именем Solution Explorer, в котором отображаются все файлы проекта в виде дерева. Эти файлы могут быть выделены в основном окне с помощью закладок с соответствующими именами. При создании проекта открывается содержимое с закладкой Form1.cs [Design]*, где расположена пустая форма с именем Form1. Пользуясь меню, можно открыть и посмотреть другие файлы проекта, которые создала VS.NET. Пока важно открыть файл с именем

Form1.cs*, содержащий программный код, уже созданный VS.NET для нашего проекта. Для этого можно щелкнуть правой кнопкой мыши на форме и в появившемся контекстном меню выбрать команду ViewCode.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WinApplAcquaintance
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

Для того чтобы полностью понять этот код, нужно познакомиться с основами объектно-ориентированного программирования, т.к. C# – объектно-ориентированный язык. Это – задача на будущее, а пока определим функции написанных предложений с точки зрения их работы в создаваемом проекте.

В первых строках программы используется ключевое слово **using**, с помощью которого к созданной программе подключаются необходимые для работы приложения библиотеки готовых подпрограмм. Это предложение не совсем точно описывает механизм работы VS.NET, но для наших первых проектов этого достаточно.

Затем в предложении `namespace WinApplAcquaintance` указывается, что все имена в данном проекте будут принадлежать пространству имен проекта `WinApplAcquaintance`.

C# является объектно-ориентированным языком, поэтому все проекты, создаваемые в нем, будут наполнены **объектами,** принадлежащими какому либо **классу**. Далее в приведенном фрагменте кода идет описание класса конкретной формы (с именем `Form1`), которую VS.NET предлагает нам использовать для создания визуальной части нашего проекта.

Дизайнерский этап работы состоит в наполнении формы нужными и удобными для работы пользователя элементами управления в соответствии с условием задачи. Для нашей простой задачи на форму достаточно поставить два управляющих элемента: метку (Label1) для помещения статического текста (Hello World! и текущая дата) и кнопку (button1), при нажатии на которую текст будет появляться на форме. Нажатие (click) на кнопку – это

некоторое событие, которое наша программа должна обработать соответствующим образом. Каждый управляющий элемент кроме свойств имеет еще набор методов – **обработчиков** некоторых событий. Их можно найти в окне Properties, если нажать кнопочку `Events`. При выборе этого обработчика VS.NET на странице программного кода создаст шаблон обработчика, в который нужно только добавить необходимые команды.

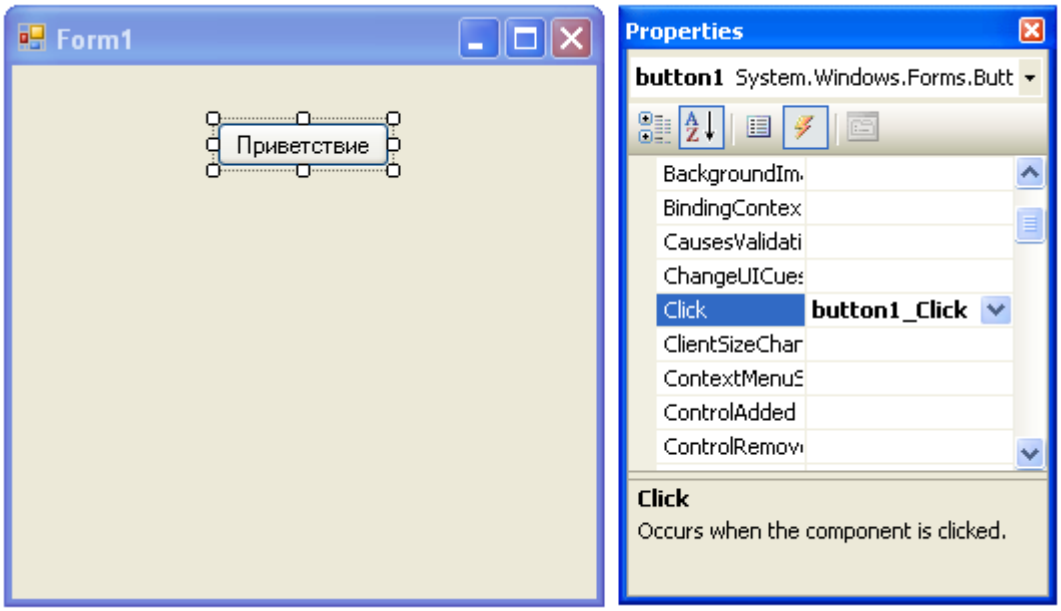


Рис. 2. Создание обработчика события

Рис. 3. Образец работающего приложения

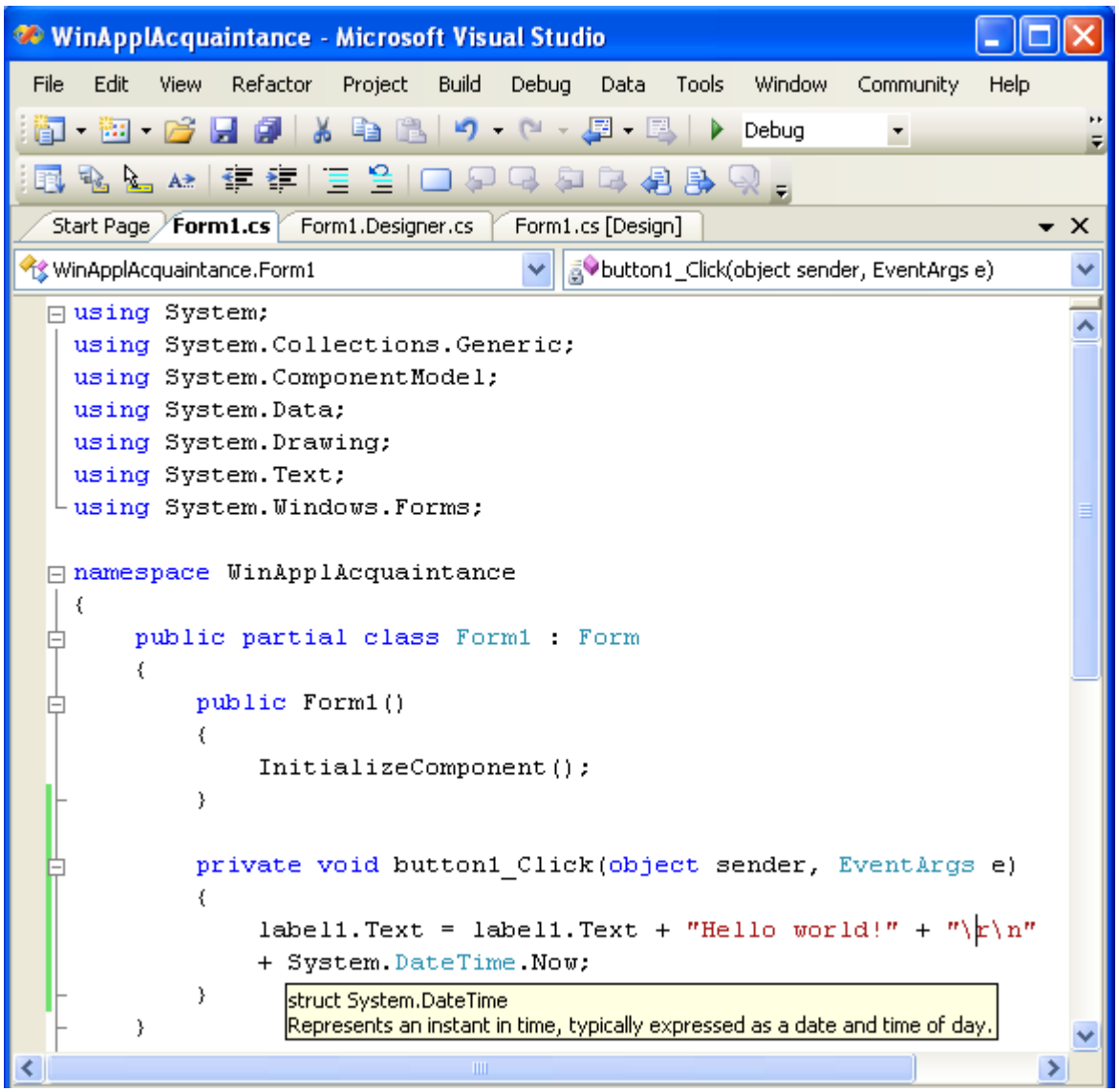


Рис. 4. Программный код приложения

Приведенный код приложения содержит всего одну процедуру – обработчик события нажатия на кнопку с именем `button1`. Этой процедуре VS.NET присвоила имя `button1_Click`. Кроме того, она снабдила ее модификатором доступа `private` (процедура может использоваться только внутри данного приложения) и служебным словом `void`, обозначающим, что процедура не возвращает никакого значения. Об аргументах, указанных в заголовке, будет рассказано позже.

Теперь рассмотрим, что же делает написанная процедура. В свойство с именем `Text` управляющего элемента `Label1` она записывает строку символов, состоящую из трех частей:

- строки “Hello World!” (значение строки заключается в двойные кавычки),
- управляющей последовательности символов “\r\n”, обозначающей вставку возврата каретки и перевода на новую строку,
- сегодняшней даты.

Все части соединяются друг с другом с помощью оператора + (конкатенации, сцепления строк). Обратите внимание, что дата получается в результате вывода значений полей структуры `System.DateTime.Now`. Редактор C# позволяет выбирать нужные свойства и методы из появляющихся контекстных меню, что очень удобно.

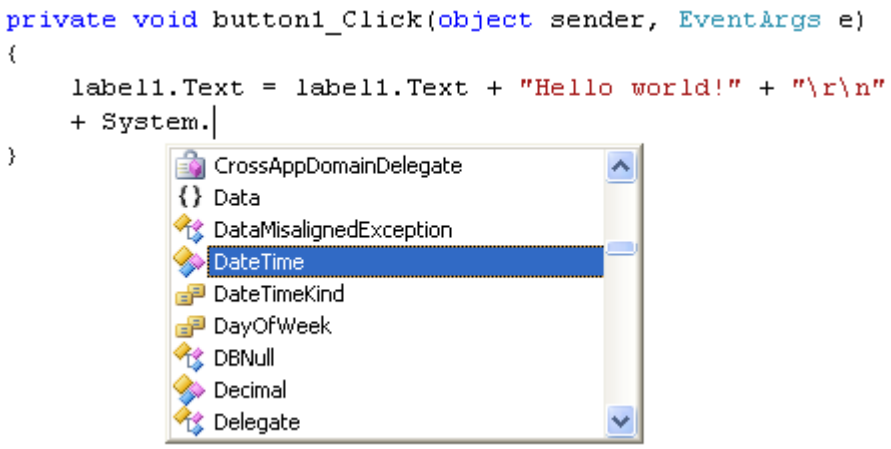


Рис. 5. Выбор имени структуры

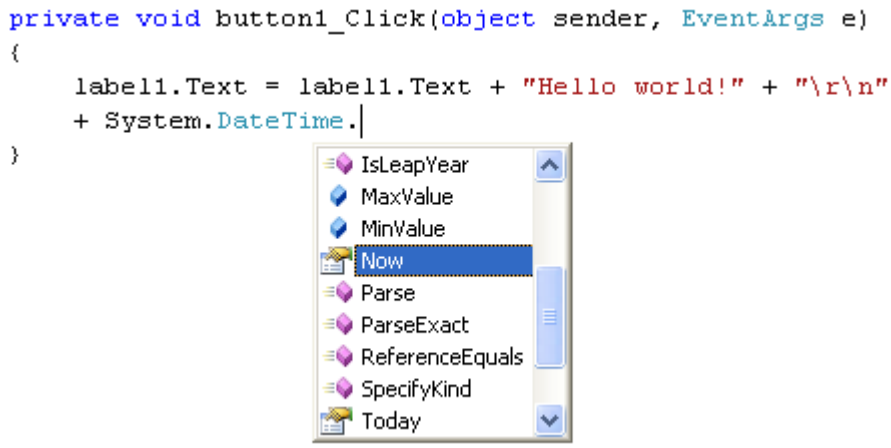


Рис. 6. Выбор свойства Now

Выполните данное задание, и приступайте к следующему. Помните, что VS.NET создает много файлов для каждого проекта. Поэтому очень важно поддерживать порядок при работе, который заключается, прежде всего, в правильном сохранении проекта. Пользуйтесь длинными именами для папок и приложений, такими, которые отражают их смысловое назначение.

На рис. 7 приведен образец формы для работающего приложения, текст задания которого помещен в свойство `Text` метки Label1.

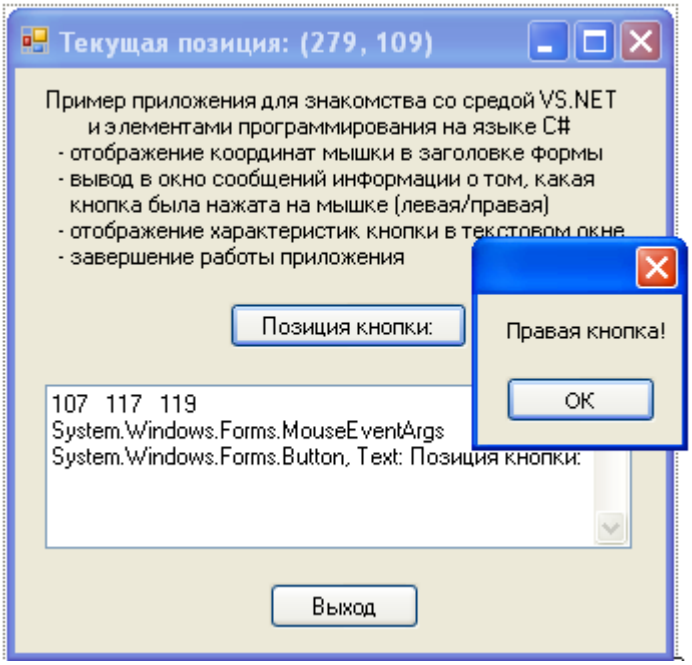


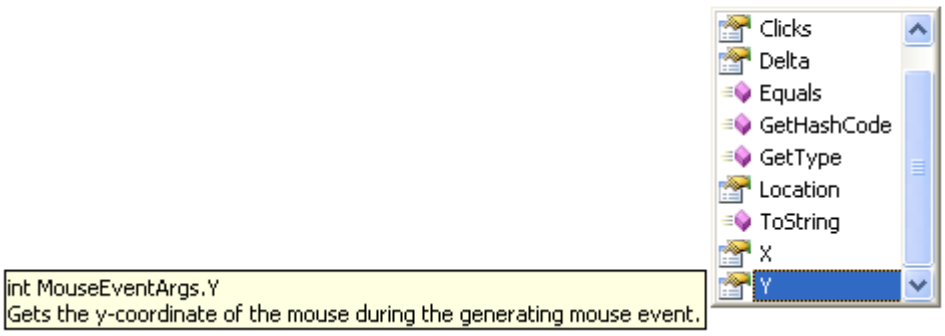
Рис. 7

На форме имеются две кнопки для двух обработчиков события нажатия на кнопку. Также видно, что информация выводится либо в стандартное окно сообщений MessageBox, либо в текстовое окно textBox1, для которого задан режим `MultiLine`. Рекомендуется изменить свойство `Name` кнопок на имена, отражающие их назначение, например, `btnClose` и `btnPosition`.

Для обработчика события нажатия на кнопку «Выход» требуется написать код, закрывающий работу приложения, выбрав соответствующий метод из выпадающего контекстного меню.

Ниже приведен код обработчика события, возникающего при движении мышки по форме, в котором координаты мышки выводятся в заголовок формы. Для этого используются данные, предоставляемые вторым параметром, указанным в заголовке обработчика, – `MouseEventArgs e`. Этими данными для мышки являются, в частности, ее координаты.

```
private void Form1_MouseMove(object sender, MouseEventArgs e)
{
    this.Text = "Текущая позиция: (" + e.X + ", " + e.      + ")";
}
```



Задание для самостоятельной работы

1. Создайте на форме метку с некоторым текстом, например, с названием задания – «Работает мышка».
2. Напишите обработчик события, когда при движении мышки над меткой, текст, расположенный в метке, принимает вид конкретного задания.
3. Напишите обработчик события, когда при движении мышки над меткой, меняются характеристики шрифта, например, его цвет.

2. Итеративное вычисление бесконечных сумм (рядов)

Задания для лабораторной работы:

1. Написать программу вычисления приближенных значений функции, представленной в виде ряда, в десяти равноотстоящих друг от друга точках указанного диапазона возможных значений аргумента.
2. Для каждого значения аргумента выполнять вычисления до достижения каждой из заданных границ точности и одновременно считать соответствующее число итераций.
3. На входе программы задать два числа: начальную и конечную точки указанного диапазона возможных значений аргумента.
4. Результаты вычислений необходимо сохранить таким образом, чтобы их можно было вывести в файл в виде приведенной ниже таблицы:

Значение аргумента *x*	Точность вычислений *eps*	Значение функции *f*	Число итераций *n*
начальное значение диапазона	10^{-1}		
	10^{-2}		
	10^{-3}		
	10^{-4}		
	10^{-5}		
	10^{-6}		
...	10^{-1}		
	10^{-2}		
	10^{-3}		
	10^{-4}		
	10^{-5}		
	10^{-6}		
конечное значение диапазона	10^{-1}		
	10^{-2}		
	10^{-3}		
	10^{-4}		
	10^{-5}		
	10^{-6}		

Как и все следующие задания, оно должно быть выполнено на языке **C#** в виде Windows-приложения.

В данных методических рекомендациях вам будет предложен вариант создания такого приложения, для чего будет представлен необходимый теоретический материал, связанный с синтаксисом языка C#, его конструкциями, классами и методами, которые нужно будет использовать при написании программного кода, а также краткое описание математической постановки задачи. Если вам не знакома математическая постановка задачи, начните изучение поставленной задачи с пояснения 3.

Пояснение 1 – понятия класса и объекта.
Пояснение 2 – работа с файлами в C#.
Пояснение 3 – определение ряда и итерационного алгоритма.
Пояснение 4 – циклические конструкции в C#.

Рекомендация 1

Начинать нужно всегда с правильного сохранения проекта, являющегося Windows-приложением, для чего следует подобрать соответствующие имена для самого проекта и для папки, в которую он будет сохранен. Для данной задачи (Итерационное вычисление бесконечных сумм) можно назвать проект WinApplLab1, а папку, например, Lab1. Рекомендуется сразу отказаться от имен, которые предлагаются по умолчанию.

Пояснение 1
Понятия класса и объекта. Создание объектов.

C# является объектно-ориентированным языком, поэтому все проекты, создаваемые в нем, будут наполнены **объектами,** принадлежащими какому-либо **классу**. Некоторые из них уже созданы авторами языка, а другие придется создавать самостоятельно. В данной лабораторной работе не потребуется создавать собственные классы. Достаточно будет тех, которые уже есть в языке. Для того чтобы понимать, как нужно работать с классами, познакомимся с некоторыми важными определениями.

1. Объявление класса

Класс определяет характеристики и поведение объектов данного класса. Его можно считать заготовкой или шаблоном, на основе которого создаются **объекты** данного класса.

В C# характеристики объектов хранятся в переменных, называемых **полями** данного класса, а моделирование поведения выполняется **методами,** которые представляют собой набор операторов, выполняющих некоторую задачу. Поля и методы называют **компонентами** класса.

Объявление класса осуществляется при помощи ключевого слова `class` и имеет следующий синтаксис:

```
[модификатор доступа] class имя-класса {тело класса}
```

Классы обычно объявляются с модификатором доступа `public`, который означает, что класс доступен везде и без ограничений.

2. Создание объекта

Создание объекта также называют **созданием экземпляра** класса, а созданный объект, соответственно, **экземпляром** класса.

Для создания объекта применяется оператор `new` и специальный метод, называемый **конструктором**.

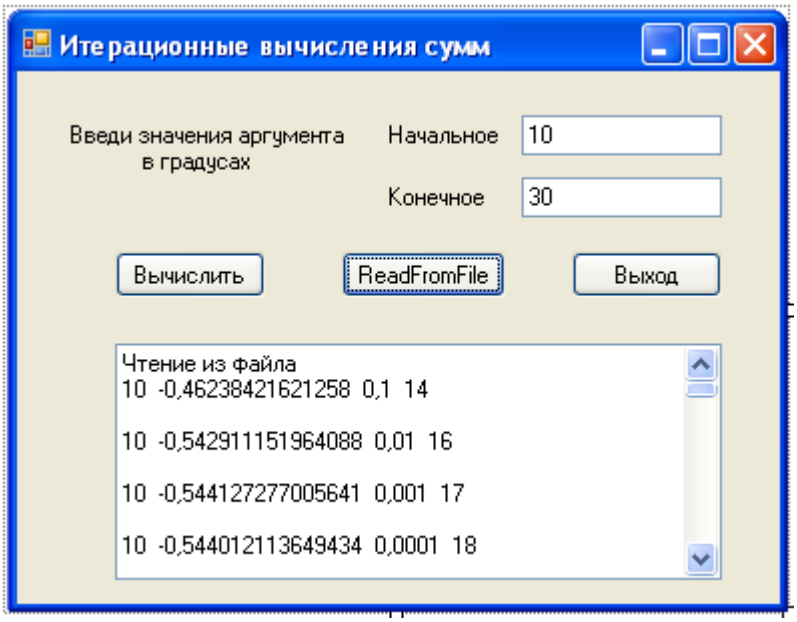
Когда вы начинаете работу с приложением в C#, язык создает для вас некоторый шаблон программного кода, в котором описаны используемые приложением программные модули, а также класс и экземпляр формы, в которой будет выполняться **визуальная** часть проекта. Посмотрите внимательно на фрагмент кода, представленный ниже.

```
//Со служебным словом using описываются модули, используемые в проекте
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WinAppLab1 //С пространством имен namespace познакомимся позднее
{
    public partial class Form1 : Form     //Объявление класса
    {
        public Form1()   //Получение экземпляра класса – нашей формы
        {
            InitializeComponent();
        }
        //Здесь пишется свой код
    }
}
```

Рекомендация 2

Никогда не рекомендуется что-нибудь менять из того, что сделал C#, можно только добавлять необходимые модули, если в этом возникает потребность.

Все программирование в C# при создании Windows-приложений состоит из двух взаимосвязанных частей – разработка визуальной части проекта и написание кода. Мы рекомендуем начинать работу с создания своего экземпляра формы – помещения на нее нужных в проекте визуальных элементов управления. Часто для них используется термин «контрол» (control).



На рисунке приведен возможный вариант формы (работающего приложения) для реализации проекта вычисления представленной в виде ряда функции **Sin x** .

На форме использованы следующие элементы управления (контролы):

- Три метки (Label1, Label2, Label3,) для вывода поясняющих заголовков («Введи значения аргумента в градусах», «Начальное», «Конечное»).
- Три элемента TextBox – два для ввода значений границ интервала (textBoxArgLeft, textBoxArgRight) и один (textBoxResult) для вывода результата вычислений. В первом случае для элементов задается режим по умолчанию (одна линия), а во втором – режим **multiLine.**
- Три кнопки (Button) для организации необходимых обработчиков событий. Для удобства чтения программного кода проекта рекомендуется давать кнопкам **символичные** имена, так как C# создает имена обработчиков с использованием имен управляющих кнопок с добавлением числового индекса (например, button1, button2). Можно выбрать имя `btnCalc` – для кнопки, нажатие на которую приводит к запуску процесса вычисления и записи результатов вычисления в файл; `btnExit` – для кнопки, нажатие на которую приводит к завершению работы проекта; `btnRead` – для кнопки, нажатие на которую приводит к чтению результатов вычислений из файла и вывода их в окно элемента `textBoxResult`.

Если вы уже программировали на каком-нибудь языке, вам придется осваивать синтаксические правила C# и отвыкать от того, к чему вы привыкли ранее.

Пояснение 2. Работа с файлами

Последнее задание лабораторной работы – запись полученных результатов в текстовый файл.

C# - язык, работающий на платформе .NET.

Любой ввод/вывод информации в .NET включает в себя использование **потоков** – абстрактных представлений устройств последовательного доступа. В качестве такого устройства может выступать расположенный на диске файл, область памяти, а также любой другой объект, который допускает последовательное считывание и запись информации.

Выходной поток используется, когда данные записываются на некоторое внешнее устройство, например, в файл на диске, как это требуется в нашей задаче. **Входной** поток используется для считывания данных в память или некоторые переменные, к которым приложение имеет доступ.

Для работы с потоками в .NET имеются специальные классы, определенные в модуле **System.IO**, который **по умолчанию не включен** в набор модулей при открытии нового проекта. Поэтому его необходимо добавить в число используемых приложением модулей, добавив инструкцию `Using System.IO` в раздел описаний стандартных модулей, используемых данным приложением.

Модуль **System.IO** содержит в себе все классы, предназначенные для считывания данных из файлов и записи данных в файлы. Для того чтобы получить в программе доступ к этим классам, необходимо сослаться на этот модуль, что и делается с помощью инструкции `using System.IO`.

Для нашей задачи достаточно классов `StreamWriter` и `StreamReader`.

Чтобы создать выходной поток – объект класса `StreamWriter`, следует воспользоваться методом-конструктором (**его имя должно совпадать с именем класса**), в котором в качестве первого параметра указать имя нужного файла, а в качестве второго – значение `true`, что будет означать, что создается новый объект.

```
StreamWriter sw = new StreamWriter("myResultFile.txt", true);
```

Для создания входного потока – объекта класса `StreamReader`, следует также воспользоваться конструктором с указанием, в качестве параметра, имени соответствующего файла.

```
StreamReader sr = new StreamReader("myResultFile.txt");
```

На приведенном рисунке формы работающего приложения видно, что результаты работы программы прочитаны из созданного текстового файла и записаны в окно элемента textbox – в его свойство `Text`.

Замечание

1. Нужно помнить, что содержимое свойства Text – это строки символов, даже если они имеют цифровое значение. Поэтому для использования в программном коде этих данных они должны быть предварительно преобразованы в необходимый тип данных. В нашем случае это делается применением метода `Convert.ToDouble`.

При написании кода метод может быть выбран из выпадающего списка для экземпляра класса.

Программный код приложения (фрагменты) с комментариями

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace WinAppIterate
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        //Обработчик события – нажатие на кнопку «Вычислить»
        private void btnCalc_Click(object sender, EventArgs e)
        {
            double S;                   //Переменная для значения суммы (функции)
            double epsilon;             //Переменная для значения точности
            double argLeft, argRight;   //Левая и правая границы  аргумента

                    . . .               //все другие необходимые объявления

            string value;               //Имя строки для вывода результата

            //Создание файла Out.txt для вывода
            StreamWriter outFile =new StreamWriter("Out.txt", true);
            //Чтение значений из элементов textBox с использованием метода Convert
            argLeft = Convert.ToDouble(textBox1.Text);
```

```
    argRight = ...
     //Вычисление диапазона изменения аргумента
    //Вычисление шага изменения аргумента
    //инициализация значения аргумента
    //цикл for  для задания одного из 5 значений аргумента
    for (i = 1; i <= 5; i++)
    {
           //Инициализация значения точности
           //Цикл for  для задания одного из 6 значений точности
           for (...)
           {
                  //Инициализация для вычислений
                  //члена ряда, суммы и числа итераций
                         ...
                  // Цикл while  вычисления суммы при
                  //заданных аргументе и точности
                  while  (Абсолютное значение члена ряда >= epsilon)
                  {
                        //Вычисление члена ряда, суммы и числа итераций
                  }
                   //Формирование строки для вывода
                   value = arg.ToString() + "   " + S.ToString() + "   "
                  + epsilon.ToString() + "   " + n.ToString() + "\r\n";
                  //Запись строки в TextBox
                  textBox3.Text = textBox3.Text+ value
                  //Запись строки в файл outFile
                  outFile.WriteLine(value);

                  //Изменение epsilon
                         ...
           }
            //Изменение аргумента
                         ...
    };
    outFile.Close();     //Так как файл в программе создается,
                          //перед каждым запуском его нужно удалять
}

//Обработчик события нажатия на кнопку "Выход"
//Завершение работы проекта
private void btnExit_Click(object sender, EventArgs e)
{
    Application.Exit();
}

//Обработчик события нажатия на кнопку "Чтение из файла"
//Создается для того, чтобы научиться читать данные из файла
private void btnRead_Click(object sender, EventArgs e)
{
    string strLine;
    StreamReader sr = new StreamReader("Out.txt");
    strLine = sr.ReadLine();
```

```
textBox3.Text = "Чтение из файла" + "\r\n";
            while (strLine != null)
            {
               textBox3.Text = textBox3.Text+strLine+ "\r\n";
               strLine = sr.ReadLine();
            }
        }
    }
}
```

Пояснение 3

Ряд – бесконечная сумма вида: $u_1 + u_2 + ... + u_n + ...$ или, короче $\sum_{n=1}^{\infty} u_n$.

Слагаемые $u_1, u_2, ..., u_n, ...$ называются членами ряда, а суммы $s_n = u_1 + u_2 + ... + u_n, \quad n = 1,2,...,$ – частичными суммами ряда порядка n.

Ряды являются важнейшим средством изображения, изучения и приближенного вычисления чисел и функций. Простейшие примеры рядов встречаются уже в элементарной математике – это бесконечные десятичные дроби, например,

$$0{,}333... = \frac{3}{10} + \frac{3}{10^2} + ...,$$

и сумма членов бесконечно убывающей геометрической прогрессии

$$1 + q + q^2 + ... + q^n + ... = \frac{1}{1-q}, \ |q| < 1.$$

Многие числа могут быть записаны в виде рядов, с помощью которых их приближенное значение вычисляется с нужной точностью. Для числа π, например, имеется ряд

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ...;$$

для числа *e*, основания натуральных алгоритмов, - ряд

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + ...;$$

для натурального логарифма *ln 2* – ряд

$$\ln = \frac{2}{3}\left(1 + \frac{1}{3}\cdot\frac{1}{9} + \frac{1}{5}\cdot\frac{1}{9^2} + \frac{1}{7}\cdot\frac{1}{9^3} + ...\right).$$

При численных расчетах ряд обычно заменяется конечной суммой *s* его первых *n* слагаемых. При этом важно иметь оценку погрешности такого вычисления (оценку «скорости сходимости» ряда) и целесообразно использовать те ряды, у которых эти погрешности достаточно быстро стремятся к нулю с ростом *n*.

Различают ряды числовые, членами которых являются постоянные числа (все приведенные выше ряды), и функциональные, членами которых являются функции, например,

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} + ... + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + ... \ .$$

Если в функциональном ряду независимым переменным придаются определенные числовые значения, то такой ряд превращается в числовой. Когда речь идет о сходимости ряда, то имеют в виду именно сходимость числового ряда.

Ниже представлены функциональные ряды для некоторых функций. Все они представляют собой **рекуррентные соотношения –** такие, которые позволяют вычислить любой член последовательности, если заданы ее первые *p* членов. Примеры наиболее известных рекуррентных соотношений:

- $a_{n+1} = qa_n \quad (q \neq 0)$ - геометрическая прогрессия;
- $a_{n+1} = a_n + d$ - арифметическая прогрессия;
- $a_{n+2} = a_{n+1} + a_n$ - последовательность чисел Фибоначчи.

ПЕРЕЧЕНЬ ФУНКЦИЙ ДЛЯ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

1. $(1+x)^{-1} = 1 - x + x^2 - x^3 + ... = \sum_{k=1}^{\infty} (-1)^{k-1} x^{k-1}; \quad -0.5 \le x \le 0.3$

2. $(1+x)^{-2} = 1 - 2x + 3x^2 - 4x^3 + ... = \sum_{k=1}^{\infty} (-1)^{k-1} k x^{k-1}; \quad -0.1 \le x \le 0.8$

3. $(1+x)^{\frac{1}{2}} = 1 + \frac{1}{2}x - \frac{1 \cdot 1}{2 \cdot 4}x^2 + \frac{1 \cdot 1 \cdot 3}{2 \cdot 4 \cdot 6}x^3 - \frac{1 \cdot 1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6 \cdot 8}x^4 + ... =$

$$= 1 + \frac{1}{2}x + \sum_{k=2}^{\infty} (-1)^{k+1} \frac{(2k-3)!!}{(2k)!!} x^k; \quad -0.15 \le x \le 0.65$$

4. $(1+x)^{-\frac{1}{2}} = 1 - \frac{1}{2}x + \frac{1 \cdot 3}{2 \cdot 4}x^2 - \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6}x^3 + ... = 1 + \sum_{k=1}^{\infty} (-1)^k \frac{(2k-1)!!}{(2k)!!} x^k;$

$$-0.25 \le x \le 0.7$$

5. $\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + 4x^4 + ... = \sum_{k=1}^{\infty} k x^k; \quad -0.7 \le x \le 0.25$

6. $e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... = \sum_{k=0}^{\infty} \frac{x^k}{k!}; \quad -7 \le x \le 5$

7. $b^x = 1 + x \ln b + \frac{(x \ln b)^2}{2!} + \frac{(x \ln b)^3}{3!} + ... = \sum_{k=0}^{\infty} \frac{(x \ln b)^k}{k!}; \quad -5 \le x \le 23$

8. $e^{-x^2} = 1 - x^2 + \frac{x^4}{2!} - \frac{x^6}{3!} + ... = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k}}{k!}; \quad -4 \le x \le 4$

9. $e^x(1+x)=1+2x+\frac{3x^2}{2!}+\frac{4x^3}{3!}+\ldots=\sum_{k=0}^{\infty}\frac{x^k(k+1)}{k!};\quad -4\le x\le 12$

10. $\sin x=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\ldots=\sum_{k=0}^{\infty}(-1)^k\frac{x^{2k+1}}{(2k+1)!};\quad -\pi\le x\le\pi$

11. $\cos x=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\ldots=\sum_{k=0}^{\infty}(-1)^k\frac{x^{2k}}{(2k)!};\quad -\pi\le x\le\pi$

12. $\operatorname{sh} x=x+\frac{x^3}{3!}+\frac{x^5}{5!}+\frac{x^7}{7!}+\ldots=\sum_{k=0}^{\infty}\frac{x^{2k+1}}{(2k+1)!};\quad -\pi\le x\le\pi$

13. $\operatorname{ch} x=1+\frac{x^2}{2!}+\frac{x^4}{4!}+\frac{x^6}{6!}+\ldots=\sum_{k=0}^{\infty}\frac{x^{2k}}{(2k)!};\quad -\pi\le x\le\pi$

14. $\sin^2 x=x^2-\frac{2^3x^4}{4!}+\frac{2^5x^6}{6!}-\frac{2^7x^8}{8!}+\ldots=\sum_{k=1}^{\infty}(-1)^{k+1}\frac{2^{2k-1}x^{2k}}{(2k)!};\quad -\frac{\pi}{2}\le x\le\frac{\pi}{2}$

15. $\cos^2 x=1-x^2+\frac{2^3x^4}{4!}-\frac{2^5x^6}{6!}+\frac{2^7x^8}{8!}-\ldots=1-\sum_{k=1}^{\infty}(-1)^{k+1}\frac{2^{2k-1}x^{2k}}{(2k)!};\quad -\frac{\pi}{2}\le x\le\frac{\pi}{2}$

16. $\ln(1+x)=x-\frac{1}{2}x^2+\frac{1}{3}x^3-\frac{1}{4}x^4+\ldots=\sum_{k=1}^{\infty}(-1)^{k+1}\frac{x^k}{k};\quad -0.86\le x\le 1.00$

17. $\ln x=(x-1)-\frac{1}{2}(x-1)^2+\frac{1}{3}(x-1)^3-\ldots=\sum_{k=1}^{\infty}(-1)^{k+1}\frac{(x-1)^k}{k};\quad 0.14\le x\le 2.00$

18. $\ln x=2\left[\frac{x-1}{x+1}+\frac{1}{3}\left(\frac{x-1}{x+1}\right)^3+\frac{1}{5}\left(\frac{x-1}{x+1}\right)^5+\ldots\right]=2\sum_{k=1}^{\infty}\frac{1}{2k-1}\left(\frac{x-1}{x+1}\right)^{2k-1};$

$0.5\le x\le 10.0$

19. $\ln x=\frac{x-1}{x}+\frac{1}{2}\left(\frac{x-1}{x}\right)^2+\frac{1}{3}\left(\frac{x-1}{x}\right)^3+\ldots=\sum_{k=1}^{\infty}\frac{1}{k}\left(\frac{x-1}{x}\right)^k;\quad 0.5\le x\le 6.0$

20. $\ln\frac{1+x}{1-x}=2\left[x+\frac{x^3}{3}+\frac{x^5}{5}+\frac{x^7}{7}+\ldots\right]=2\sum_{k=1}^{\infty}\frac{1}{2k-1}x^{2k-1};\quad -0.70\le x\le 0.46$

21. $\ln\frac{x+1}{x-1}=2\left[\frac{1}{x}+\frac{1}{3x^3}+\frac{1}{5x^5}+\ldots\right]=2\sum_{k=1}^{\infty}\frac{1}{(2k-1)x^{2k-1}};\quad -8\le x\le -2$

22. $\ln\frac{x}{x-1}=\frac{1}{x}+\frac{1}{2x^2}+\frac{1}{3x^3}+\ldots=\sum_{k=1}^{\infty}\frac{1}{kx^k};\quad 2\le x\le 8$

23. $\ln\frac{1}{1-x}=x+\frac{x^2}{2}+\frac{x^3}{3}+\ldots=\sum_{k=1}^{\infty}\frac{x^k}{k};\quad -0.40\le x\le 0.72$

24. $\frac{1-x}{x}\ln\frac{1}{1-x}=1-\frac{x}{2}-\frac{x^2}{6}-\frac{x^3}{12}-\ldots=1-\sum_{k=1}^{\infty}\frac{x^k}{k(k+1)};\quad -0.54\le x\le -0.15$

25. $\arcsin x = x + \frac{1}{2\cdot 3}x^3 + \frac{1\cdot 3}{2\cdot 4\cdot 5}x^5 + \frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6\cdot 7}x^7 + ... = \sum_{k=0}^{\infty}\frac{(2k)!}{2^{2k}(k!)^2(2k+1)}x^{2k+1};$

$-0.29 \le x \le 0.81$

26. $\operatorname{arctg} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + ... = \sum_{k=0}^{\infty}\frac{(-1)^k x^{2k+1}}{2k+1}; \quad -1 \le x \le 1$

27. $\operatorname{arctg} x = \frac{x}{\sqrt{1+x^2}}\left[1 + \frac{1}{6}\left(\frac{x^2}{1+x^2}\right) + \frac{3}{40}\left(\frac{x^2}{1+x^2}\right)^2 + ...\right] =$

$= \frac{x}{\sqrt{1+x^2}}\sum_{k=0}^{\infty}\frac{(2k)!}{2^{2k}(k!)^2(2k+1)}\left(\frac{x^2}{1+x^2}\right)^k; \quad -17 \le x \le 17$

28. $\operatorname{arctg} x = \frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \frac{1}{7x^7} - ... = \frac{\pi}{2} - \sum_{k=0}^{\infty}(-1)^k\frac{1}{(2k+1)x^{2k+1}}; \quad 1 \le x \le 20$

29. $(\arcsin x)^2 = x^2 + \frac{x^4}{3} + \frac{2^4 x^6}{3\cdot 5\cdot 6} + ... = \sum_{k=0}^{\infty}\frac{2^{2k}(k!)^2 x^{2k+2}}{(2k+1)!(k+1)}; \quad -0.94 \le x \le 0.33$

30. $\operatorname{arcsec} x = \frac{\pi}{2} - \frac{1}{x} - \frac{1}{2\cdot 3x^3} - \frac{1\cdot 3}{2\cdot 4\cdot 5x^5} - ... = \frac{\pi}{2} - \sum_{k=0}^{\infty}\frac{(2k)!x^{-(2k+1)}}{2^{2k}(k!)^2(2k+1)};$

$1.6 \le x \le 10.0$

В качестве примера рассмотрим задачу вычисления приближенного значения функции

$$f(x) = x + \frac{x^2}{2} + \frac{x^3}{3} + ... + \frac{x^n}{n} = \sum_{k=1}^{n}\frac{x^k}{k}.$$

Введем обозначение: $a_k = \frac{x^k}{k}$ – k-е слагаемое. Тогда начальный элемент последовательности слагаемых $a_1 = x$, т.е. для номеров выполняется условие $1 < k \le n$. Найдем зависимость, связывающую между собой два соседних слагаемых. Запишем выражение для (k-1)-го слагаемого $a_{k-1} = \frac{x^{k-1}}{k-1}$. Выразим a_k через a_{k-1}: $a_k = a_{k-1}\cdot\frac{x(k-1)}{k}$. Получилось рекуррентное соотношение для последовательности слагаемых:

$a_1 = x$,

$a_k = a_{k-1}\cdot\frac{x(k-1)}{k}, \quad k > 1.$

Назовем сумму первых k слагаемых S_k частичной суммой. Для последовательности частичных сумм тоже записывается рекуррентное соотношение:

$S_1 = x,$

$S_k = S_{k-1} + a_k, \quad 1 < k \le n.$

Оба рекуррентных соотношения можно использовать в алгоритме одновременно, в одном цикле.

Рекуррентные соотношения являются основой **рекуррентных или итеративных алгоритмов**, смысл которых заключается в постепенном приближении вычисляемого значения к результату. Итерация – это многократно повторяющееся действие. Каждый шаг выполнения итеративного алгоритма соответствует одной итерации.

Выделяют два варианта задач работы с рекуррентными последовательностями:

1) вычислить значение элемента последовательности с заданным номером;
2) выполнить последовательные вычисления до достижения заданной точности.

Для функций, которые могут быть представлены в виде бесконечной суммы (в виде ряда), слагаемые связаны друг с другом рекуррентными соотношениями. В таких рядах выполняется условие сходимости – при добавлении очередного слагаемого к частичной сумме она постепенно приближается к истинному значению функции для заданного значения аргумента. Необходимым условием этого является уменьшение абсолютного значения слагаемого с увеличением его номера. Когда абсолютное значение очередного слагаемого становится меньше заданной точности, вычисления можно завершить и считать, что значение функции получено с этой самой точностью.

Основой итеративного алгоритма является цикл.

Пояснение 4
Циклические конструкции в C#

1. Цикл **for**

Общий формат этого цикла, предназначенного для повторного выполнения одной инструкции, имеет вид:

for (*инициализация; условие; итерация*) инструкция;

Общий формат цикла *for*, предназначенного для повторного выполнения программного блока, имеет вид:

```
for (инициализация; условие; итерация )
{
        последовательность инструкций
}
```

Элемент «инициализация» обычно представляет собой инструкцию присваивания, которая устанавливает управляющую переменную цикларавной начальному значению. Эта переменная действует в качестве счетчика, который управляет работой цикла.

Элемент «*условие*» представляет собой выражение типа `bool`, в котором проверяется значение управляющей переменной цикла. Результат проверки определяет, выполнится цикл `for` еще раз или нет.

Элемент «*итерация*» - это выражение, которое определяет, как изменяется значение управляющей переменной цикла после каждой итерации.

Элементы в заголовке цикла отделяются друг от друга точкой с запятой. Цикл `for` будет выполняться до тех пор, пока вычисление элемента *условие* дает истинный результат. Как только *условие* станет ложным, выполнение программы продолжится с инструкции, следующей за циклом `for`. Управляющая переменная цикла `for` может изменяться как с положительным, так и с отрицательным приращением, причем величина этого приращения может быть любой. Вот некоторые примеры:

```
//Вывод значений счетчика на экран в диапазоне от 100 до -95
for (int x = 100; x > -100; x -= 5) Console.WriteLine(x);
for (int count = 10; count < 5; count++)
     int a += count; // Эта инструкция не будет выполнена
```

Цикл `for` особенно полезен в тех случаях, когда известно количество его повторений.

В цикле `for` допускается использование нескольких управляющих переменных цикла. В этом случае инструкции инициализации и итерации для каждой из этих переменных отделяются **запятыми.**

```
//Вывод значений счетчиков на экран
for (int i = 0, int j = 10; i < j; i++, j--)
   Console.WriteLine("i и j: " + i + " " + j);
```

В разделах инициализации и итерации можно использовать любое количество инструкций, но обычно их число не превышает двух.

Условным выражением, которое управляет циклом `for`, может быть любое выражение типа `bool`. Например, в следующем примере цикл управляется переменной `done`.

```
int i, j;
bool done = false;
for (i = 0, j > 100; !done; i++, j--)
{
     if (i * i >= j) done = true;
}
Console.WriteLine("i,j: " + i + "  " + j);
```

В `C#` разрешается опустить любой элемент заголовка цикла (инициализация, условие, итерация) или даже все сразу. Это может дать интересный результат. Рассмотрим примеры.

```
int i;
for (i = 0; i < 10;)
{
     Console.WriteLine (“Шаг № ” +i);
     i++;
}
```

Здесь отсутствует выражение итерации цикла `for`. Вместо него увеличение на 1 переменной `i` выполняет инструкция, находящаяся внутри цикла. Цикл проработает нормально – выведет номера шагов по порядку.

В следующем примере в заголовке цикла только условие.

```
int i;
i = 0; //инициализация переменной цикла до заголовка цикла
for (; i < 10;)
{
     Console.WriteLine (“Шаг № ” +i);
     i++; //Увеличение переменной цикла в теле цикла
}
```

Бесконечный цикл

```
for (;;)
{
     //Программисты так иногда делают.
     //Цикл будет работать без конца.
     //Завершить такой цикл можно с помощью инструкции break
}
```

Циклы без тела

В C# тело, связанное с циклом `for` (или каким-нибудь другим циклом), может быть пустым. Пустая инструкция синтаксически допустима, и она часто оказывается полезной. Например, следующий «бестелесный» цикл позволяет получить сумму чисел от 1 до 5.

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

Процесс суммирования выполняется внутри заголовка цикла и поэтому тела цикла просто не нужно.

Следует обратить внимание на итерационное выражение: `sum += i++`. Такие инструкции весьма распространены в C#; они легко понимаются, если их разбить на части. В данном случае она означает, что в переменную `sum` необходимо поместить результат сложения текущего значения переменной `sum` и значения переменной `i`, а затем увеличить `i` на 1.

В приводимых выше примерах переменная i (счетчик цикла) иногда объявлялась внутри заголовка цикла, а иногда – вне его. В первом случае эта переменная объявлялась как локальная для цикла и переставала существовать после завершения цикла. Во втором случае ее значением можно пользоваться, если это требуется, где-то далее в программе.

Цикл **while**

Общая форма цикла while имеет такой вид:

```
while условие) инструкция;
```

Под элементом *инструкция* понимается либо одиночная инструкция, либо блок инструкций. Работой цикла управляет элемент условие, который представляет собой любое допустимое выражение типа bool. Цикл работает, пока условное выражение возвращает значение true. Как только оно становится ложным, управление передается инструкции, которая следует за циклом.

Следующий цикл может быть применен для вычисления порядка (order) целого числа (num).

```
int num = 435679;
int ord = 0;
while (num > 0)
{
     ord++;
     num = num / 10;
}
```

2. Цикл **do while**

В отличие от циклов for и while, в которых условие проверяется при входе в цикл, цикл do while проверяет условие при выходе из цикла. Это значит, что цикл do while всегда выполняется хотя бы один раз. Его общий формат имеет такой вид:

```
do
{
     инструкции
} while (условие);
```

Фигурные скобки желательно писать всегда для улучшения читабельности цикла do while, даже если тело цикла состоит из одной инструкции. Тем самым не допускается путаница с циклом while.

Следующий фрагмент программы позволяет вывести на экран целое число с обратным порядком цифр.

```
int  num = 198;
int nextDigit;
Console.WriteLine (“Число: ” + num);
Console.WriteLine (“Число с обратным порядком цифр: ”);
```

```
do
     {
          nextDigit = num % 10;
          Console.WriteLine (nextDigit);
          num = num / 10;
     } while (num > 0);
```

3. Цикл **foreach**

Этот цикл предназначен для опроса элементов **коллекции,** к которым относится **массив.** Он будет рассмотрен позже.

С циклами часто используются инструкции `break`, `continue` и `return`.

С помощью инструкции `break` можно организовать немедленный выход из цикла, опустив выполнение кода, оставшегося в его теле и проверку условного выражения. Управление передается инструкции, следующей за циклом.

С помощью инструкции `continue` можно организовать «досрочный» выход из текущей операции цикла. Она принудительно выполняет переход к следующей итерации, опуская выполнение оставшегося кода в текущей. Инструкцию `continue` можно рассматривать как дополнение к более «радикальной» инструкции `break`.

Инструкция return обеспечивает возврат из метода с возвращением вычисленного в методе значения.

3. Перевод чисел из одной системы счисления в другую

Задания для лабораторной работы:
Написать программу, которая вводит:

1) основание системы счисления p, 2<= p <=10000;
2) последовательность цифр этого числа. Каждая цифра – десятичное число, в последовательности эти цифры разделяются пробелами;
3) новое основание системы счисления q, 2<= q <=10000.

Результат работы программы – последовательность цифр представления числа по новому основанию. Каждая цифра – десятичное число, в последовательности эти цифры разделяются пробелами.

Замечание.

Значение вводимого числа не должно превышать максимально допустимого предела для внутреннего представления целочисленных данных.

Задание, должно быть выполнено на языке **C#** в виде Windows-приложения.

В данных методических рекомендациях вам будет предложен вариант создания такого приложения, для чего будет представлен необходимый теоретический материал, связанный с синтаксисом языка C#, его конструкциями, классами и методами, которые нужно будет использовать при написании программного кода, а также краткое описание математической постановки задачи. Если вам не знакома математическая постановка задачи, начните изучение поставленной задачи с пояснения 3.

Пояснение 1 – Массивы в C#.
Пояснение 2 – Строки в C#.
Пояснение 3 – Перевод числа из одной системы счисления в другую.

Рекомендация 1

Начните с сохранения приложения. Рекомендуется сразу отказаться от имен, которые предлагаются по умолчанию. Для данной задачи («Перевод числа из одной системы счисления в другую») можно назвать приложение WinApplLab2, а папку, например, Lab2. Можно сделать имена соответствующими смыслу приложения, например, назвать папку LabPerevod, а приложение – WinApplPerevod.

В данном приложении входная информация (разряды числа) будет представлена в виде строки символов, представляющей собой цифровые значения и пробелы. Эта строка должна будет подвергаться некоторой обработке для превращения ее в массив числовых значений. Поэтому ниже приводятся свойства и методы **строк** и **массивов**.

Пояснение 1. Массивы в C#

В языке C# массив – это тип, производный от класса `System.Array`.

Формальное определение обычного массива (в C# есть и другие) выглядит как обычно: массив – это набор элементов, доступ к которым осуществляется посредством числового индекса. Массивы могут содержать элементы любого встроенного типа данных C#. Массивы могут быть простыми и многомерными. Они объявляются путем помещения квадратных скобок ([]) после типа данных для элементов этого массива. В данной задаче потребуется работа только с одномерными массивами. Массив символьных строк с 10 элементами можно объявить следующим образом:

```
string [] value;
value = new string [10];
```

Такое объявление требует две строки кода. Однако те же действия можно задать и в одной строке:

```
string [] value = new string [10];
```

В любом случае для создания массива фиксированного размера необходимо использовать ключевое слово `new` (конструктор по умолчанию). Нужно помнить, что в C# при создании массива всем его элементам присваивается значение по умолчанию в зависимости от типа элементов. Например, для массива целых чисел всем элементам будет изначально присвоено значение 0, а все элементы строкового массива получат значение пустой строки.

Так как все массивы в C# являются производными от класса `System.Array`, любой из них наследует от базового (родительского) класса большое количество полезных методов и свойств. В таблице приведены некоторые из них.

Clear()	Этот статический метод дает возможность очистить указанный диапазон элементов (числовые элементы приобретут значение 0, а ссылки на объекты - null)
CopyTo()	Копирование элементов из исходного массива в массив назначения
GetLength()	Метод GetLength() используется для определения количества элементов в указанном измерении массива
Length	Length – это свойство только для чтения, с помощью которого можно получить количество элементов массива
GetLowerBound()	Метод применяется для получения нижней границы заданного измерения массива
GetUpperBound()	Метод применяется для получения верхней границы заданного измерения массива
GetValue()	Возвращает значение элемента массива для указанного индекса
SetValue()	Устанавливает значение элемента массива для указанного индекса
Reverse()	Этот статический метод позволяет расставить элементы одномерного массива в обратном порядке
Sort()	Сортирует одномерный массив встроенных типов данных.

Пояснение 2. Строки (string) в C#

В программах часто приходится использовать последовательности символов. Для этой цели применяются **строки**, в которых можно хранить последовательности символов Unicode. Стандарт Unicode систематизирует представления символов большинства языков, существующих в мире. Для хранения одного символа отводится 16 бит.

Строки в C# являются объектами класса System.String. В этом классе определены свойства и обширный набор методов для работы со строками.

Length – общедоступное свойство класса String, возвращающее количество символов в строке.

Ниже в таблице содержится описание некоторых наиболее часто используемых методов класса String.

Метод	Тип возвращаемого значения	Описание
Concat() (статический)	string	Перегружен. Возвращает результат конкатенации двух или более строк. Возвращаемая строка строится путем добавления каждой следующей строки к концу предыдущей. Перегруженный оператор сложения (+) также выполняет конкатенацию строк.
Copy() (статический)	string	Возвращает новую строку, которая является копией данной. Перегруженный оператор присваивания (=) также корирует строки.
Equals() (статическая версия и версия реализации)	bool	Перегружен. Возвращает, булево значение, указывающее, равны ли две строки. Перегруженный оператор равенства (==) также проверяет строки на равенство.
CopyTo()	void	Копирует указанное число символов строки в указанную позицию в массиве символов.
Insert()	string	Вставляет строку в указанную строку в указанном месте.
Split()	string[]	Перегружен. Разбивает строку на массив строк, используя указанный разделитель. Возвращает полученный массив строк.
Substring()	string	Перегружен. Возвращает подстроку данной строки.
ToCharArray()	Char[]	Перегружен. Копирует символы строки в указанный массив символов.
ToString()	string	Перегружен. Возвращает значение строки.
Trim()	string	Удаляет пробелы или указанные символы в начале или конце строки.

По условию задачи второй лабораторной работы требуется при вводе данных отрабатывать неправильные действия пользователя – набор неверного символа при вводе последовательности цифр для оснований систем счисления и значений разрядов числа, а также не учет граничных значений – 2 и 10000 для оснований систем счисления. Так как значения вводятся из компонентов textbox, они имеют тип `string`, и придётся воспользоваться некоторыми методами и свойствами этого типа:

1. Свойство `Length`,
2. Метод `Equals()` или перегруженный оператор равенства (==)

Для разделения строки - последовательности разрядов заданного числа на массив подстрок можно воспользоваться методом `Split()`.

Объявление строки и присваивание ей значения текста из компоненты `textBox2`:

```
string sr = textBox2.Text;
```

Присваивание некоторой стринговой переменной значения i-го элемента строки `sr`:

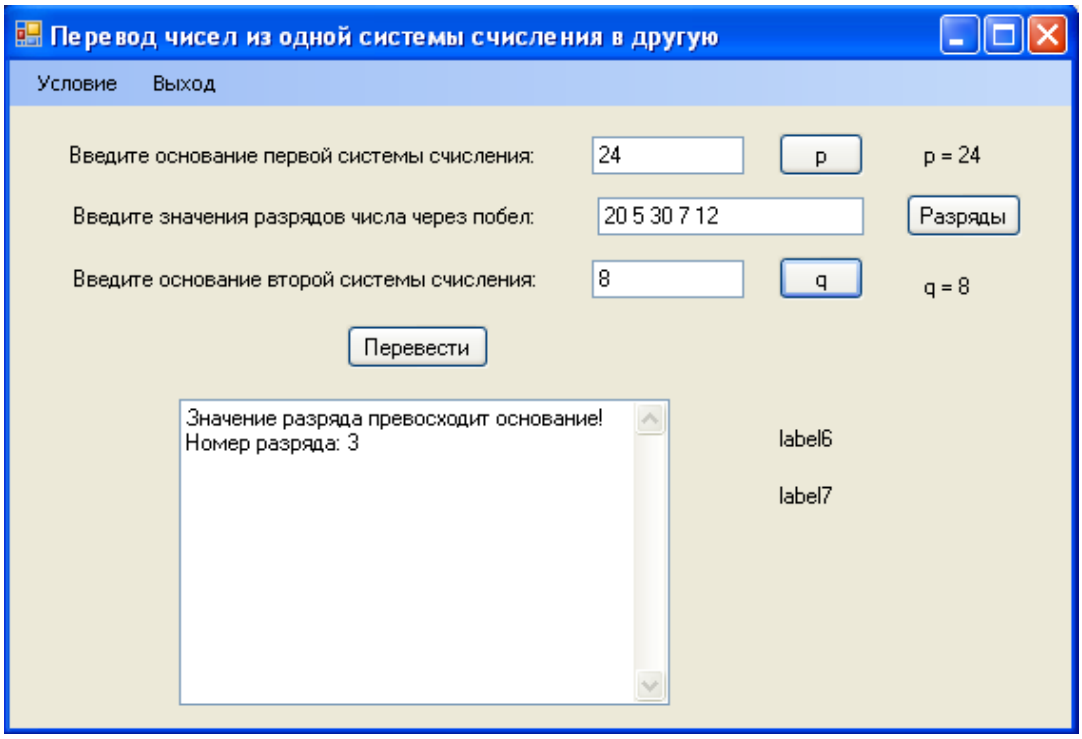
```
string c = sr[i].ToString();
```

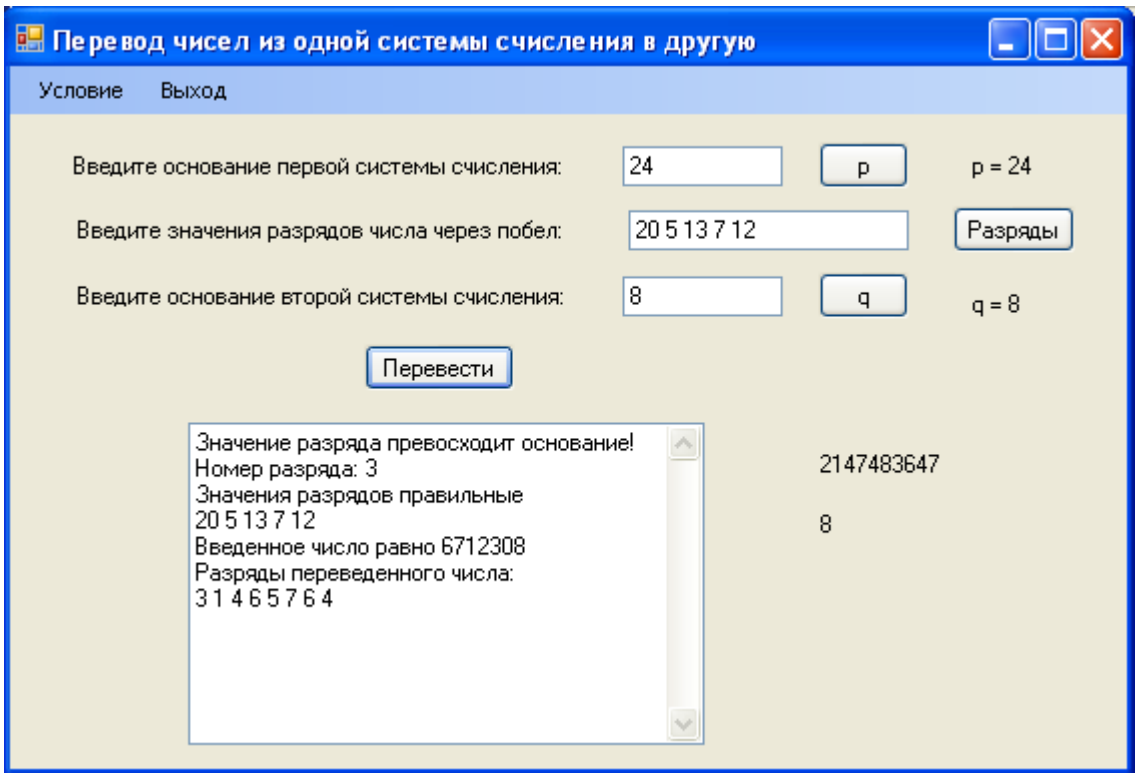
Сравнение значения переменной `c` со значениями констант – символов десятичных цифр:

```
... && (!(c.Equals("8"))) && (!(c=="9"))) ...
```

Рекомендация 2

Мы рекомендуем начинать работу с создания своего экземпляра формы – помещения на нее нужных в проекте визуальных элементов управления. На рисунках приведен возможный вариант формы работающего приложения для двух случаев – когда ввод данных произведен с ошибкой и когда приложение работает верно.





На форме использованы следующие элементы управления (контролы):

- Несколько меток для отображения поясняющих заголовков, введенных значений p и q, значения максимального целого числа и пр.
- Четыре элемента TextBox - два для ввода значений и для вывода результата вычислений. В первом случае для элементов задается режим по умолчанию (одна линия), а во втором – режим **multiLine.**
- Четыре кнопки (Button) с именами «p», «q», «Разряды», «Перевести» для организации необходимых обработчиков событий. Для удобства чтения программного кода проекта рекомендуется давать кнопкам **символичные** имена.
- Управляющий элемент для создания **меню.**

При написании кода обработчиков нужно обязательно учесть обработку ошибочных действий пользователя:

- Ввод не цифрового символа,
- Ввод значений оснований систем счисления, выходящих за заданные в условии задачи пределы.
- Нарушение правил ввода значений разрядов числа при заданной системе счисления.
- Ввод разрядов целого числа, приводящих к получению значения, превышающего максимальное целое число, представимое в C# (переполнение).

Программный код приложения (фрагменты) с комментариями

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

namespace WinAppLabSistShisl
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    private void выходToolStripMenuItem_Click
                                    (object sender, EventArgs e)
        {
            Close();
        }

     int p = 0, q = 0;                  //Основания систем счисления
     string sr;                          //строка последовательности разрядов
     string[] numStr =new string[1000];   //строковый массив для выделения
                                        //строк-значений разрядов из входной строки
     int[] number = new int[1000];  //числовой массив значений разрядов
                                        //во входном числе
     int kolNum = 0;                    //количество разрядов во входном числе
     string value;                  //строка для вывода результата в окно TextBox
     int lengthr;                    //длина строки последовательности разрядов
     int m;                          //число разрядов в результирующем массиве
     int res;                        //переменная для функции DivRem
     int[] mNum=new int[10000];      //массив разрядов числа-результата
     int er = 0;                         ////индикатор наличия переполнения
     Boolean flagError = false;     //для фиксации ошибок в записи разрядов
     int valNum;                        // вспомогательная переменная

    //Ввод основания p (нажатие на соответствующую кнопку)
     private void btn_P_Click(object sender, EventArgs e)
     {
         string sp = textBox2.Text;  //Читаем строку для основания . p

          //В цикле от 0 до длины строки проверяем, является ли читаемый символ
          //не символом цифры. Если это так, в стандартное окно сообщений выводим
          //строку «Введен неверный символ» и с помощью инструкции return
          //возвращаемся к  строке ввода для исправления ошибки
          int i = 0;
```

```
            while (i < sp.Length)
            {
                string c = sp[i].ToString();
                if ((!(c.Equals("0"))) && (!(c.Equals("1"))) &&
                   (!(c.Equals("2")))&&(!(c.Equals("3")))&&
                   (!(c.Equals("4")))&&(!(c.Equals("5"))&&
                   (!(c.Equals("6")))&&(!(c.Equals("7")))&&
                   (!(c.Equals("8")))&& (!(c=="9")))
                {
                   MessageBox.Show("Введен неверный символ");
                   return;
                }
                i++;
            }
            //Если все символы для  p  введены верно, проверяем находится ли p в
            //нужном числовом диапазоне. НАПИШИТЕ КОД ДЛЯ ЭТОЙ ПРОВЕРКИ!

                        • • •
        }
//--------- Конец обработчика для ввода основания системы счисления p.--------

        //Ввод разрядов числа в первой системе счисления   p
        //Обработчик нажатия на кнопку
        private void btn_razr_Click(object sender, EventArgs e)
        {
            sr = textBox3.Text;  //Читаем строку разрядов из textBox3
            //В цикле проверяем правильность введения символов
                            • • •
            //Сначала превращаем  последовательность разрядов заданного числа в
            //массив подстрок numStr, используя метод Split
            // с разделителем-пробелом.
            numStr = sr.Split(Convert.ToChar(" "));

            //В  цикле foreach каждую подстроку проверяем на правильность ввода.
            //Значение разряда не может быть больше основания системы счисления
              foreach (string lin in numStr)
              {
                  kolNum += 1;
                  if (Convert.ToInt32(lin) >= p)
                  {
                      flagError = true; //Фиксируем ошибку
                      textBox1.AppendText
                       ("Значение разряда превосходит основание!
                    Номер разряда: "+ kolNum.ToString() + "\r\n");
                    kolNum = 0;
                    break;
                  }
              }
              string value = "";
              if (flagError == false) //Если нет ошибок в записи разрядов
              {
                  textBox1.AppendText("Значения разрядов" +
                                      "правильные" + "\r\n");
```

```
kolNum = 0;

                    //В цикле foreach заносим значения в числовой массив разрядов
                    //number[] для введенного числа и считаем число разрядов kolNum

                                        . . .
                    //Выводим разряды в textBox1.

                                        . . .
            }
            flagError = false;   //логической переменной возвращаем
                                 // начальное значение
        }

//------------ Конец обработчика для ввода значений разрядов.-------------


    //Ввод основания системы счисления q (нажатие на соответствующую кнопку)
    private void btn_q_Click(object sender, EventArgs e)
    {
                        . . .
    }

//--------- Конец обработчика для ввода основания системы счисления  q.--------

    //Перевод числа из системы счисления p в систему счисления q
    private void btnPerevod_Click(object sender, EventArgs e)
       {
            //Сначала проверка на переполнение при вычислении десятичного
            //значения введенного числа по формуле (схеме) Горнера
            valNum = number[0];
            for (int k = 1; k < kolNum; k++)
            {
                if ((Int32.MaxValue - number[k]) / p < valNum)
                {
                    er = 1; //Индикатор наличия переполнения
                    MessageBox.Show("Error! OverFlow!");
                    break;
                }
                else { valNum = valNum * p + number[k]; }
            }

            if (er == 0)
            {
                textBox1.Text = textBox1.Text +
                "Введенное число равно " + valNum.ToString() + "\r\n";
                label6.Text = Int32.MaxValue.ToString();
            }

          //Перевод числа в систему с другим основанием - q
           //Используем функцию DivRem(valNum, q, out res), которая
           //вычисляет результат целочисленного деления valNum на q и в
           //переменной out res возвращает остаток от целочисленного деления
```

```
• • •
        }
//---Конец обработчика для перевода числа из одной системы счисления в другую.---


      private void условиеToolStripMenuItem_Click
                                (object sender, EventArgs e)
      {
          MessageBox.Show ("Оформить текст задания");
      }
   }
}
```

Пояснение 3
Перевод чисел из одной системы счисления в другую

Для разработки алгоритма перевода определим информационную модель представления числа в системе счисления с основанием **p**. Число **A** в обычной десятичной системе счисления записывается как последовательность цифр: $\mathbf{a_n a_{n-1} \ldots a_1 a_0}$. Его значение равно:

$$\mathbf{A = a_n 10^n + a_{n-1} 10^{n-1} + \ldots + a_1 10^1 + a_0 10^0}.$$

В системе счисления с основанием **p** используются **p** цифр: 0, 1, 2, … p-1. Число **B**, записанное в этой системе в виде: $\mathbf{b_n b_{n-1} \ldots b_1 b_0}$, имеет значение

$$\mathbf{B = b_n p^n + b_{n-1} p^{n-1} + \ldots + b_1 p^1 + b_0 p^0}.$$

На практике иногда применяют системы счисления с основанием, отличным от 10. Так, например, время в секундах и минутах записывается в шестидесятеричной системе счисления. Компьютеры выполняют вычисления в самой простой двоичной системе. В ней используются только две цифры – 0 и 1. Если в системе счисления основание меньше 10, то в качестве цифр применяют начальные десятичные цифры от 0 до 9. Если же **p** > 10, то первые буквы латинского алфавита a, b,… служат цифрами 10, 11, … . Основание системы счисления записывают как индекс числа, например:

$$1018_{10} = 3FA_{16} = 1272_8 = 1111111010_2.$$

Будем считать, что цифры числа **A** записываются в элементы массива **mass**: младшая цифра – в **mass[0]**, следующая – в **mass[1]**, и т.д. до **mass[n]**. Рассмотрим две задачи:

1) Заданы **p**, **mass**, и **n**; вычислить **A**.
2) Заданы **p** и **A**; вычислить **mass** и **n**.

Для упрощения алгоритма преобразуем вышеприведенную формулу для вычисления значения числа в схему Горнера:

$$\mathbf{A = (\ldots(mass[n]p + mass[n-1])p + \ldots + mass[1])p + mass[0]}.$$

В этой формуле мы избавились от степеней p. Теперь легко записать оператор цикла, решающий задачу 1 из приведенного ниже задания:

Написать программу, которая переводит целое положительное число, заданное в **p**-ичной системе счисления в **q**-ичную систему счисления. Для ввода чисел в системе счисления с основанием **p** использовать цифры от 0 до **p-1** и вводить их через пробел. В конце ввода пробел не ставить.

Для вывода чисел в **q**-ичной системе счисления использовать цифры от 0 до **q-1** и выводить их тоже через пробел.

Вычисление числа по его представлению в массиве.

Введем названия переменных, которые будут использоваться далее в программе. Пусть число называется **numIn**, а соответствующий ему массив разрядов **mass.** Как видно из приведенной схемы Горнера, младший разряд числа записывается в нулевой элемент массива mass, а старший – в последний (n-й).

Вычисление значения числа, заданного в системе счисления p, будет выполнено с помощью следующего **downto** цикла:

```
numIn := 0;
for i := n downto 0 do
  numIn := numIn * p + mass[i]
```

Задание: расписать цикл по шагам для заданного значения n и значений элементов массива mass.

Для того чтобы написать алгоритм перевода числа, заданного в одной системе счисления в другую, придется воспользоваться операциями целочисленного деления **div** (вычисляет результат целочисленного деления) и **mod** (вычисляет остаток от целочисленного деления).

Рассмотрим пример: пусть задано число **A** в 8-ичной системе счисления. Значения его разрядов от нулевого по второй (2 5 6) Требуется определить значения его разрядов при записи в 5-ичной системе счисления.

Шаг 1: вычисляем значение этого числа по приведенному выше алгоритму, т.е. по формуле:

$$A=6 * 8^2 + 5 * 8^1 + 2 * 8^0 = 426$$

Шаг 2: выполняем последовательное целочисленное деление полученного числа на новое основание системы счисления, например, 5. Получаем два значения – результат целочисленного деления и остаток от целочисленного деления. Продолжаем это деление до тех пор, пока результат целочисленного деления не станет <= 0.

426 div 5 = 85 и 426 mod 5 = 1
85 div 5 = 17 и 85 mod 5 = 0
17 div 5 = 3 и 17 mod 5 = 2
3 div 5 = 0 и 3 mod 5 = 3

Полученное число разрядов в пятеричном представлении числа – 4. Сами разряды, начиная с младшего – 1 0 2 3.

Как видно, значения разрядов – это последовательно получаемые остатки. Можно снова по схеме Горнера вычислить значение числа.

A= 3 * 5^3 + 2 * 5^2 + 0 * 5^1 + 1 * 5^0 = 375 + 50 + 0 + 1 = 426

Если полученное значение равно ранее полученному значению, перевод прошел правильно.

```
n:=0;
while numIn > 0 do
begin
   massNew[n]:=numIn mod q;
   numIn:=numIn div q;
   n:=n+1;
end;
```

Массивы – параметры методов

В C# определено несколько способов работы с последовательностями данных одного типа. В лабораторной работе «Перевод чисел из одной системы счисления в другую» уже использовались **массивы** данных, для которых было характерно то, что количество элементов в них не изменялось в процессе работы программы. Такие массивы называют **статическими**. При объявлении массива используется следующий синтаксис:

```
type[] array_name;
```

Здесь `type` – тип данных для элементов массива, а `array_name` –имя, назначаемое массиву. Квадратные скобки после имени типа означают, что объявляется массив, а не отдельная переменная или объект. Приведенное выражение только **объявляет** массив, а далее в программе еще необходимо **создать** массив с определенным количеством элементов. Для создания массива в C# используется оператор `new`. Например, в следующем примере создается статический массив с именем `int_Array` из 100 элементов типа `int`.

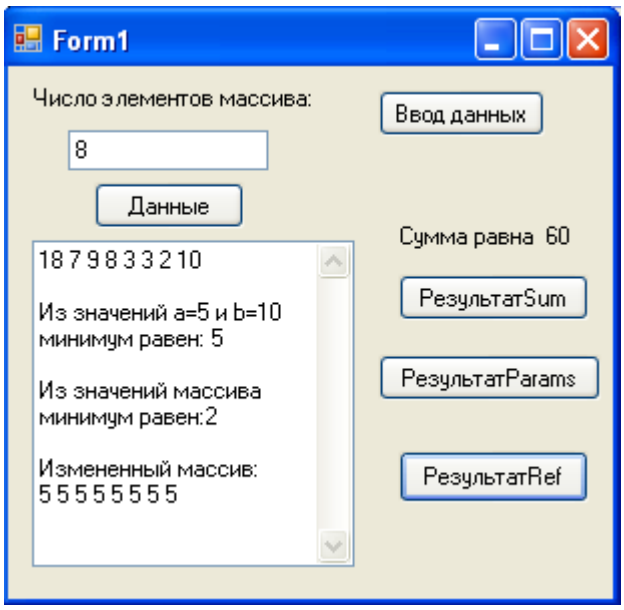
```
int[] intArray;
int_Array = new int[100];
```

Массивы в C# являются объектами класса **System.Array**. В этом классе определено много полезных свойств и методов для работы с массивами. Например, элементы массива можно упорядочить, пользуясь имеющимся в классе методом `Sort()`. Имя массива, который должен быть упорядочен, передается этому методу в качестве **параметра**. Методы в большинстве случаев используются с параметрами. В C# применяются атрибуты для параметров методов. Они называются модификаторами. Перечень модификаторов приведен в нижеследующей таблице.

Модификатор параметра	Назначение
`(нет)`	Если параметр записан без модификатора, то по умолчанию считается, что это входной параметр, передаваемый в метод в качестве копии исходного значения (передача параметра по значению).
`out`	Выходной параметр, который возвращается вызванным методом вызывающему.
`ref`	Входное значение присваивается вызывающим методом, но оно может быть изменено в вызванном методе. Передается ссылка на параметр (передача по ссылке).
`params`	Этот модификатор позволяет передавать целый набор параметров как единое целое. В любом методе может быть только один модификатор `params`, и параметр с этим модификатором может быть только последним параметром в списке параметров.

Целью лабораторной работы № 3 является изучение различных алгоритмов сортировки массивов целочисленных переменных. Массивы в этой работе можно рассматривать как статические массивы. Однако, лучше сразу написать все алгоритмы в виде **процедур** (методов), при оформлении которых массивы, подлежащие упорядочению, передаются методу в качестве **параметров**.

Ниже приведен вариант формы для работающего приложения с использованием процедур с различными способами передачи параметров, а следом – программный код приложения с комментариями, которые помогут разобраться в соответствующих механизмах оформления и работы методов с использованием массивов – параметров с различными атрибутами. Приложение написано для простых алгоритмов – вычисления суммы элементов массива, изменения элементов массива, генерирования массива с заданным числом элементов, нахождения минимального элемента массива.




```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WinAppOpenArray
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        int len;        //Переменная для числа элементов в массиве
        int[] myData;   //Объявление массива
    //---------------------------------------------------------------------

        //Процедура для генерирования массива значений.
        //Сгенерированный массив является выходным параметром поцедуры
        //и объявляется с атрибутом out
        private int[] DataInput(int lenData, out int[] mas)
        {
            mas = new int[len];
            Random num = new Random();
            for (int i = 0; i < len; i++)
            {
                mas[i] = num.Next(20);
            };
            return mas;
        }

    //---------------------------------------------------------------------

        //Процедура для вычисления суммы элементов массива с входными
        //параметрами: Data (значения элементов массива)
        //l (число элементов массива) и выходным параметром result (out)
        private void summa(int[] Data, int l, out int result)
        {
```

```
        result = 0;
        for (int i = 0; i < l; i++)
        {
            result = result + Data[i];
        }
    }


    //-----------------------------------------------------------------

    //Процедура, изменяющая значения элементов массива, выбранного по
    //ссылке.Параметр процедуры, представляющий массив, имеет атрибут ref.
    private void Change(ref int[] RefData)
    {
        int l = RefData.GetUpperBound(0);     //определение верхнего
                        //граничного значения числа элементов массива,
                        //на который указывает ссылка
        //В цикле все элементы массива могут быть изменены
        for (int i = 0; i <= l; i++)
        {
            RefData[i] = 5;
        }
    }


    //-----------------------------------------------------------------

    //Процедура с атрибутом params для параметра, возвращающая
    //минимальный из введеных значений, в том числе и минимальный
    //элемент массива значений (см. ниже обработчик btnParams_Click).
    private int minVal(params int[] numbers)
    {
        int min;
        min = numbers[0];
        for (int i = 1; i < numbers.Length; i++)
            if (numbers[i] < min)
                min = numbers[i];
        return min;
    }


//---------------Обработчики событий нажатия на кнопки ----------------

    //Создание массива целых чисел - обработчик нажатия на кнопку «Данные»
    private void btnGenerate_Click(object sender, EventArgs e)
    {
        if (textBox2.Text == "")
        {
            MessageBox.Show("Введите число элементов!");
        }
        else
        {
            len = Convert.ToInt32(textBox2.Text);
            myData = new int[len];
            Random num = new Random();
            for (int i = 0; i < len; i++)
            {
                myData[i] = num.Next(20);
            }
        }
        string value = "";
        for (int i = 0; i < len; i++)
        {
            value = value + myData[i].ToString() + " ";
```

```
            }
            textBox1.Text = value + "\r\n" + "\r\n";
        }

        //Обработчик события - обращение к процедуре вычисления
        //суммы элементов массива (кнопка «РезультатSum»)
        private void btnSum_Click(object sender, EventArgs e)
        {
            int result; //Этот параметр должен быть указан с описателем out
                        //в списке параметров при вызове процедуры
            summa(myData, len, out result);
            lbSum.Text = "Сумма равна  " + result.ToString();
        }


        //Обработчик события - обращение к процедуре изменения массива,
        //выбранного по ссылке (кнопка «РезультатRef»)
        private void btnChange_Click(object sender, EventArgs e)
        {
            Change(ref myData);
            string value = "";
            for (int i = 0; i < len; i++)
            {
                value = value + myData[i].ToString() + " ";
            }
            textBox1.Text = textBox1.Text + "Измененный массив:" + "\r\n";
            textBox1.Text = textBox1.Text + value;
        }

        //Обработчик события - обращение к процедуре с атрибутом параметра
        //params  для вычисления минимального их входных значений.
        private void btnParams_Click(object sender, EventArgs e)
        {
            int a=5; int b=10;
            int minNum = minVal(a, b);
            textBox1.Text = textBox1.Text + "Из значений a=5 и b=10 +
                  минимум равен: " + minNum.ToString() + "\r\n" + "\r\n";
            minNum = minVal(myData);
            textBox1.Text = textBox1.Text + "Из значений массива минимум +
                   равен:" + minNum.ToString() + "\r\n" + "\r\n";
        }
        //обработчик события нажатия на кнопку «Ввод данных» для получения
        //массива значений с помощью процедуры DataInput
        private void btnInput_Click(object sender, EventArgs e)
        {
            if (textBox2.Text == "")
            {
                MessageBox.Show("Введите число элементов!");
            }
            else
            {
                len = Convert.ToInt32(textBox2.Text);
                DataInput(len, out myData);
                string value = "";
                for (int i = 0; i < len; i++)
                {
                    value = value + myData[i].ToString() + " ";
                }
                textBox1.Text = value + "\r\n" + "\r\n";
            }
        }
    }
}
```

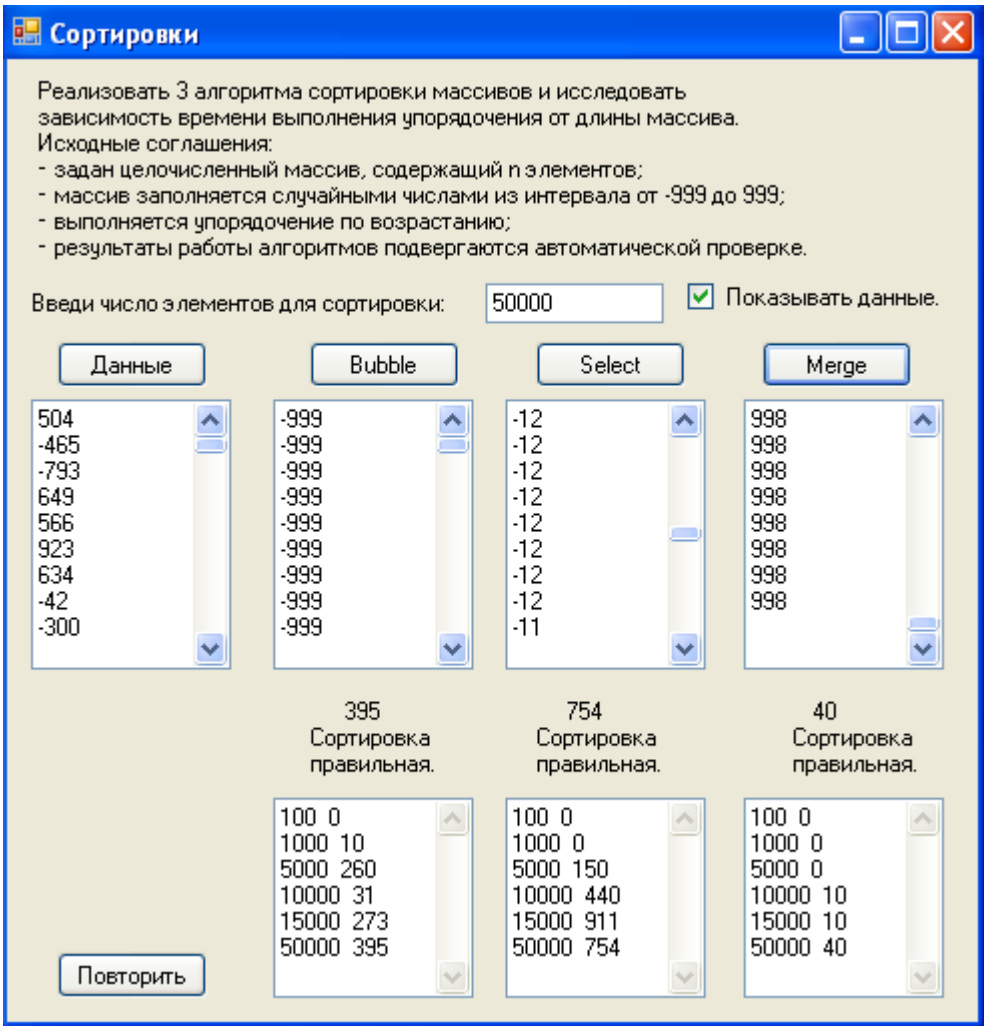
4. Алгоритмы сортировки на языке C#

Задания для лабораторной работы:

1. Изучить способ получения последовательности случайных чисел в языке C# (класс `Random` и его методы).
2. Создать форму проекта для реализации алгоритмов:
   - обменной сортировки,
   - сортировки выбором,
   - сортировки вставками,
   - сортировки слиянием (рекурсивный вариант).

3. Добавить в проект функцию проверки упорядоченности и выполнить проверку работы каждого алгоритма. Предусмотреть вывод на форму результата проверки.

4. Для каждого из реализованных алгоритмов сортировки составить таблицу зависимости времени выполнения от длины `n` упорядочиваемого массива. Рекомендуемые значения `n`: 10, 100, 1000, 10000, 100000.



На приведенном выше рисунке показан возможный вид формы работающего приложения для заданий 1 – 4.

Текст задания отображается в свойстве Text метки (Label1) при наведении на нее курсора мышки (для мышки используются методы MouseMove и MouseLeave).

Входные данные и результаты сортировок выводятся в элементы TextBox, для которых задан режим MultiLine.

Для ввода числа элементов в сортируемом массиве используется элемент TextBox в режиме одной строки.

Элемент CheckBox (Показывать данные) добавлен на форму для выбора режима вывода данных в элементы TextBox.

Пояснение к выполнению задания 1 лабораторной работы

Для генерирования входных данных нужно воспользоваться имеющимся в C# классом Random и правилами, которые приняты в этом языке для создания объектов класса. Объекты создаются на основе типов, так же, как и переменные. В объектно-ориентированном программировании тип объекта известен под названием «класс».

Ниже приведен пример кода с комментариями, который можно использовать для создания последовательности случайных значений и занесения их в нужный массив.

```
     //Читаем значение размерности len массива из TextBox
 len = Convert.ToInt32(textBoxInp.Text);
      //Создаем экземпляр массива с размерностью l
  MyData = new int[len];
int number;
int i;
Random num;  //Описываем объект класса Random
     //с помощью оператора new создаем объект
     //и инициализируем его по умолчанию
 num = new Random();
      //В цикле заполняем массив случайными числами из
      //требуемого диапазона, используя метод Next
for (i = 0; i < len; i++)
{
  number=num.Next(-100, 100);
  MyData[i] = number;
};
```

Пояснение к выполнению задания 4 лабораторной работы

В C# есть возможность представления ДАТЫ и ВРЕМЕНИ в программе. Дата и время в программе представляются при помощи

структуры System.DateTime. Эта структура содержит много свойств и методов для манипуляции с датами и временем.

Кроме того в программе могут быть представлены промежутки времени, известные как ВРЕМЕННЫЕ ИНТЕРВАЛЫ.

Для представления временных интервалов используется класс System.TimeSpan (span – диапазон). Этот класс содержит много методов для манипуляций с промежутками времени.

Ниже приведен пример программы (консольное приложение), в которой измеряется время, необходимое для сложения большого количества чисел.

Возьмите его за образец написания кода при определении временных интервалов работы алгоритмов сортировки.

```
using System;
class Example
{
    public static void Main();
    {
    //Создание объекта DateTime и присвоение ему значения текущего времени
        DateTime start = dateTime.Now;
        //Сложение чисел в цикле for
        long total = 0;
        for (int count = 0; count < 1000000; count++)
        {
            total += count;
        }
        //Вычитание текущего времени из значения переменной start,
        //разница хранится в экземпляре класса TimeSpan
        TimeSpan timeTaken = DateTime.Now - start;
        //вывод числа миллисекунд, потраченных на сложение чисел
       Console.WriteLine("Milliseconds = " +
                            timeTaken.Milliseconds);

        //Вывод суммы всех чисел
        Console.WriteLine("total = " + total);
    }
 }
 //-------------------------------------------------------------------
 //Эта программа выводит следующие строки:
      Milliseconds = 10
      total = 455555500000
```

Для сравнительного анализа времени работы алгоритмов можно воспользоваться свойством **Environment.TickCount.**

Переведите нижеследующий текст и воспользуйтесь предложенным примером в своей программе.

Gets the number of milliseconds elapsed since the system started.

// Sample for the Environment.TickCount property.

```
// TickCount cycles between Int32.MinValue, which is a negative
// number, and Int32.MaxValue once every 49.8 days. This sample
// removes the sign bit to yield a nonnegative number that cycles
// between zero and Int32.MaxValue once every 24.9 days.

using System;
    int result = Environment.TickCount & Int32.MaxValue;
```

This example produces the following results:

```
TickCount: 101931139
```

`TickCount` ПОВТОРЯЕТСЯ между `Int32.MinValue`, которое является отрицательным числом, и `Int32.MaxValue` каждые 49.8 дней. Этот образец удаляет бит знака, чтобы давать число nonnegative (не отрицательное число), которое повторяется между нулем и `Int32.MaxValue` каждые 24.9 дней.

Рекомендация 1

При большом числе элементов входного массива для определения времени работы алгоритмов сортировки **не следует** выводить данные в элементы `TextBox`. Этого можно достигнуть снятием флажка в управляющем элементе `CheckBox` (Показывать данные).

Рекомендация 2

При большой размерности сортируемого массива для увеличения скорости вывода значений в компоненты `TextBox` рекомендуется сначала из всех элементов массива вместе с символами перехода на новую строку (`"\r\n"`) сформировать единую строку (переменную типа `string`), которую затем использовать для вывода.

Пояснение к выполнению задания 3 лабораторной работы

Поскольку проверка правильности выполнения сортировки должна в программе выполняться несколько раз, требуется оформить это действие в виде функции, к которой можно было бы обращаться неоднократно. В C# для этого нужно оформить соответствующий метод.

Любой метод содержит одну или несколько инструкций. В хорошей C# программе один метод выполняет только одну задачу. Каждый метод имеет **имя** и именно это имя используется для его **вызова**. В качестве имен методов нельзя использовать зарезервированные в C# слова для выполнения служебных функций и ключевые слова.

Формат записи метода такой:

```
Модификатор_доступа тип_возврата имя (список_параметров)
{
        //тело метода
}
```

Модификатор доступа определяет, какие части программы могут получить доступ к методу. Если он не указан, подразумевается, что метод закрыт (`private`) в рамках класса, где он определен. Если метод определен с модификатором `public`, то его могут вызывать все остальные составные части программного кода, даже те, которые определены вне класса, в котором метод объявлен.

С помощью описателя `тип_возврата` указывается тип значения, возвращаемого методом. Это может быть любой допустимый тип, включая типы классов, создаваемые программистом. Если метод не возвращает никакого значения, нужно указать тип `void`.

Элемент `список_параметров` представляет собой последовательность пар (состоящих из типа данных и идентификатора), разделенных запятыми. Параметры – это переменные, которые получают значения аргументов, передаваемых методу при вызове. Если метод не имеет параметров, `список_параметров` остается пустым.

Чтобы добавить метод в класс, необходимо определить его внутри объявления класса.

В общем случае существует два варианта условий для возвращения из метода. Первый определяется обнаружением закрывающей скобки для тела метода. Второй вариант состоит в выполнении инструкции `return`. Эта инструкция обычно используется для методов, возвращающих значения. Возвращаемое значение может являться результатом вычислений, может означать, успешно или нет выполнены действия, составляющие метод, или представлять собой некий код состояния.

Методы возвращают значения вызывающим их процедурам, используя следующую форму инструкции `return`:

```
return значение;
```

Важно помнить, что тип значения, возвращаемого методом, должен быть совместимым с типом возвращаемого значения, указанного в заголовке определения метода.

Ниже приведен код метода (с именем `fun_check`), реализующего проверку правильности выполнения сортировки массива целых чисел, возвращающего логическое значение `true`, если сортировка выполнена верно и `false` – в противном случае. Проверка выполняется по следующему алгоритму:

Поскольку в данной работе используется целочисленный массив, содержащий значения только из определенного интервала (от -999 до 999), можно подсчитать, сколько раз каждое значение встречается в исходном и упорядоченном массивах. Сравнение счетчиков позволит убедиться в правильности упорядочения.

```
//Проверка правильности сортировки
private Boolean fun_check(int[] a, int[] aSort)
      {
         //Цикл для проверки упорядоченности по возрастанию массива aSort
         int i = 0;
         while (i < kol - 2 & aSort [i] <= aSort [i + 1])
            i++;
         if (i < kol - 2)
            return false;    //Если упорядоченность не обнаружена
         else
         {
            //Создаем массивы-счетчики значений
            int[] ma = new int[1998];
            int[] maSort = new int[1998];

            //Следующий цикл не обязателен, т.к. при создании
            //массивы инициализирутся нулями.
            for (i = 0; i < 1998; i++)
            {
               ma[i] = 0;
               maSort [i] = 0;
            };
             //Увеличиваем на 1 соответствующие значения
             //элементов массивов-счетчиков.
             //Массивы-счетчики нумеруются с 0.
            for (i = 0; i < kol; i++)
            {
               ma[a[i] + 999]++;
               maSort[aSort [i] + 999]++;
            };
            //Сравниваем попарно все значения массивов-счетчиков
            i = 0;
            while (i < 1997 & ma[i] == maSort [i])
               i++;
            //Формируем возвращаемые логические значения-результаты
            if (i == 1997)
               return true;
            else
               return false;
         }
      }
```

5. Построение графиков функций

При решении научных, учебных и деловых программ часто требуется построить графики различных функций. Функции `y = f(x)`, где «x» – независимая переменная (аргумент) и «y» – зависимая переменная (метод) могут быть заданы в программе различным образом. Это может быть некая формула (аналитический способ), например, `y = sin(x)`. Другой способ состоит в том, что переменные «x», «y» могут быть заданы таблично, иными словами, в виде массивов `x(i)` и `y(i)`, где `i` – 0, 1, 2, …, `Npoints` (число точек графика в этом случае равно `Npoints + 1`, а число отрезков между точками графика равно `Npoints`).

В данной лабораторной работе предлагается построить график для функций, заданных таблично.

Задания для лабораторной работы:

1. Используя данные, полученные при выполнении первой лабораторной работы и записанные в файл, построить графики зависимости числа итераций от точности вычисления суммы бесконечного ряда для всех значений аргумента.
2. Данные, записанные в файл, предварительно нужно прочитать в таблицу, из которой затем производить выбор нужных значений – аргумента, погрешности вычислений и числа итераций.

Для выполнения данной лабораторной работы необходимо будет познакомиться с элементом `PictureBox`, который в C# используется для отображения графики, с его свойствами и методами.

Поскольку требуется построить график для функции, заданной в виде таблицы, нужно будет также изучить способы работы с таблицами в C# – классом `DataGridView`, с его свойствами и методами.

Работа с таблицей

Ниже приводится ряд рекомендаций для оформления проекта, аналогичного заданию лабораторной работы. Для работы с ним необходимо создать произвольный текстовый файл, содержащий табличную информацию о функции (значение аргумента и значение функции) и поместить его в папку для лабораторной работы. Данные в файле должны представлять собой набор строк, и значения в строке должны разделяться одним пробелом. Можно выполнить проект с этим искусственно созданным файлом, а можно сразу (по аналогии) приступить к оформлению проекта для лабораторной работы № 4.

1. Создайте форму, на которую поместите следующие компоненты:
- таблицу DataGridView
- TextBox в режиме multiLine
- 2 кнопки с именами:
  btnFileToGrid – для чтения данных из файла в таблицу,
  btnGridToTextBox – для чтения данных из таблицы.
- управляющий элемент OpenFileDialog для удобства выбора нужного файла с набором данных.

2. Таблица DataGridView имеет свойства Columns и Rows, представляющие наборы ее столбцов и строк, которые нумеруются с нуля. Открыв в окне Properties для свойства Columns редактор (при нажатии на знак многоточия), создайте два столбца для значений аргумента и значения функции соответственно.
Введите для их свойств HeaderText и Name следующие значения:

	0-й столбец	1-й столбец
HeaderText	Аргумент	Функция
Name	ColumnArg	ColumnFun

Разумеется, при выполнении лабораторной работы № 4 следует создать таблицу с 4 столбцами и с соответствующими названиями.

3. Не забудьте, что файл с данными желательно разместить в той же папке, что и проект. Щелкнув по кнопке btnFileToGrid, получите шаблон обработчика события для чтения данных из файла в таблицу. Напишите этот обработчик, используя следующий алгоритм:
- Проверить, правильность открытия файла, выбранного в диалоговом окне:
```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
```
- Создать экземпляр класса StreamReader.
- В цикле (пока не дошли до конца файла) считать последовательно все строки из файла, выполнив для них следующие преобразования.
- Каждую считанную строку преобразовать в массив подстрок с помощью метода Split для переменных типа String. Эти подстроки и будут представлять собой значения в ячейках (столбцах) для каждой строки в таблице. В нашем примере столбцов в каждой строке – 2, а для файла лабораторной работы № 4 – четыре.
- Чтобы записать эти значения в таблицу, нужно воспользоваться классом C#, который представляет собой коллекцию строк для класса DataGridView. Это класс DataGridViewRowCollection, который имеет метод, позволяющий добавлять строки к коллекции, которым мы и воспользуемся.

Ниже показано создание экземпляра этого класса (с именем rows), который связывается со строками таблицы dataGridView с помощью указателя this.

```
DataGridViewRowCollection rows = this.dataGridView.Rows;
```

Написанное выше предложение основано на том, что в C# все экземпляры (объекты) классов являются ссылочными типами. Поэтому оно означает, что, выполняя действия над объектом rows, мы обращаемся по ссылке к нашей таблице – ее свойству Rows. Это обеспечивает неявная переменная this, представляющая собой указатель на тот экземпляр класса, который был использован при вызове метода.

Метод, позволяющий добавлять строки к коллекции, которым можно воспользоваться для записи преобразованных строк из файла в таблицу, имеет следующий вид: `rows.Add(row);`

После выполнения этих действий таблица окажется **заполненной** данными из файла.

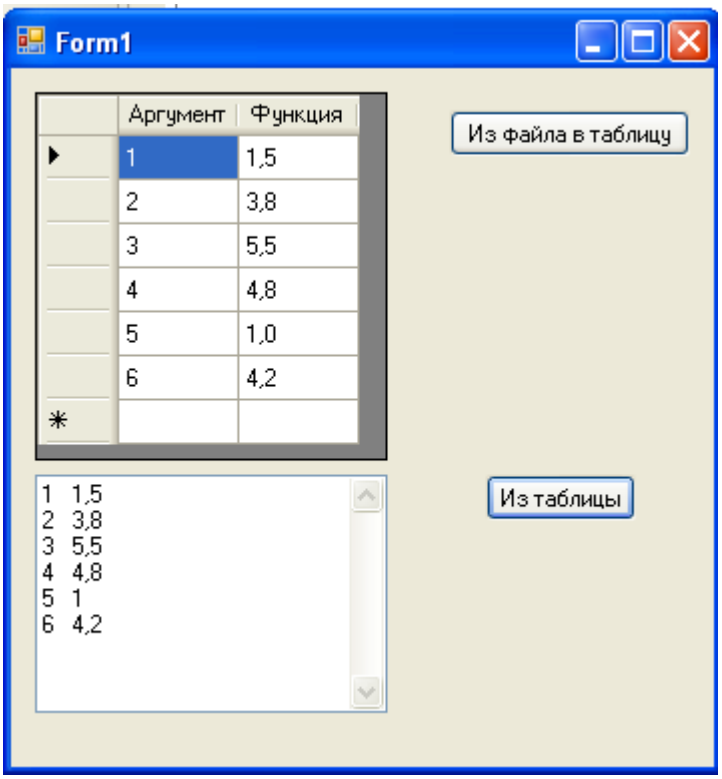
Теперь нужно научиться **выбирать** данные из таблицы. В предлагаемой задаче данные должны быть записаны в поле TextBox. Для выполнения этих действий предназначен второй обработчик события – нажатия на кнопку btnGridToTextBox.

Что нужно помнить?

Таблица – это набор строк. Как уже было сказано выше, они нумеруются с 0. С другой стороны, таблица – это набор ячеек. Для каждой строки каждая ячейка (cell) задается номером столбца, которые тоже нумеруются с 0. Например, для i-й строки таблицы DataGridView нашего примера обращение к нулевому и первому столбцу i-ой строки будет выглядеть так:

```
double arg =
     Convert.ToDouble(DataGridView.Rows[i].Cells[0].Value);
double fun =
     Convert.ToDouble(DataGridView.Rows[i].Cells[1].Value);
```

В нашей задаче (построение графика зависимости) данные, хранящиеся в ячейках таблицы, для вывода в поле TextBox должны быть интерпретированы в строковый тип. На рисунке приведен пример работающего приложения для примера.



Работа с графикой

Следующим этапом работы является построение графика функции, для значений, заданных в таблице. Для этого на форму нужно поместить Control-элемент `PictureBox`. Для того чтобы этот элемент визуально выделялся на области формы, рекомендуется сначала на форму поместить элемент-контейнер `Panel`, у которого задать свойство `BorderStyle` равным `FixedSingle`, а уже потом поместить на него – `PictureBox`. В классе `PictureBox` определено важное свойство – `Graphics`. С помощью многочисленных свойств класса `Graphics` можно выводить в область, занятую элементом управления, текстовые надписи, геометрические фигуры и различные изображения.

Класс `Graphics` передается в качестве параметра `PaintEventArgs e` для события `Paint`, которое играет очень важную роль в приложениях `Windows Forms`. Оно обеспечивает **перерисовку** окна приложения, как только в этом возникает необходимость, например, если поле одного окна было временно занято другим окном, или требуется вывод другого графика.

Поэтому задание построения графика рекомендуется выполнять, вызывая метод `Paint`.

Ниже приведен фрагмент кода для вывода изображения системы координат с использованием метода DrawLine (рисовать линию) класса Graphics.

Сначала объявляется класс Graphics для элемента PictureBox1 с именем grpBox, задается цвет пера boxPen - черный и толщина пера – 2 и затем рисуются две перпендикулярные линии. Линии рисуются с использованием экземпляров класса Point (точка). Точка задается двумя координатами с помощью оператора new. В определении координат используются свойства Width (ширина) и Height (высота) элемента PictureBox1.

```
private void pictureBox1_Paint(object sender, PaintEventArgs e)
{
   Graphics grpBox = e.Graphics;
   Pen boxPen = new Pen(Color.Black, 2);
   grpBox.DrawLine(boxPen,
                new Point(10, 10), new Point(10, pictureBox1.Height - 10));
   grpBox.DrawLine(boxPen,
              new Point(10, pictureBox1.Height - 10),
             new Point(pictureBox1.Width - 10, pictureBox1.Height - 10));
}
```

Рассмотрим подробнее вопрос о том, как были выбраны координаты точек – начала и конца прямых для координатных осей.

Построение графиков на экране монитора имеет следующие важные особенности.

Первая особенность заключается в том, что интерфейс графического устройства выводит график (по умолчанию) в системе координат, начало которой расположено в верхнем левом углу области рисования. Ось x направлена в этом случае вправо, а ось y – вниз. Когда, например, мы будем размещать график в элементе управления pictureBox, то начало координат будет размещено именно в левом верхнем углу этого элемента. Для нашей задачи (число итераций и значения точности – положительные величины) начало системы координат нужно разместить в левом нижнем углу. Поэтому начало системы координат будет в точке:

point (10, pictureBox1.Height - 10).

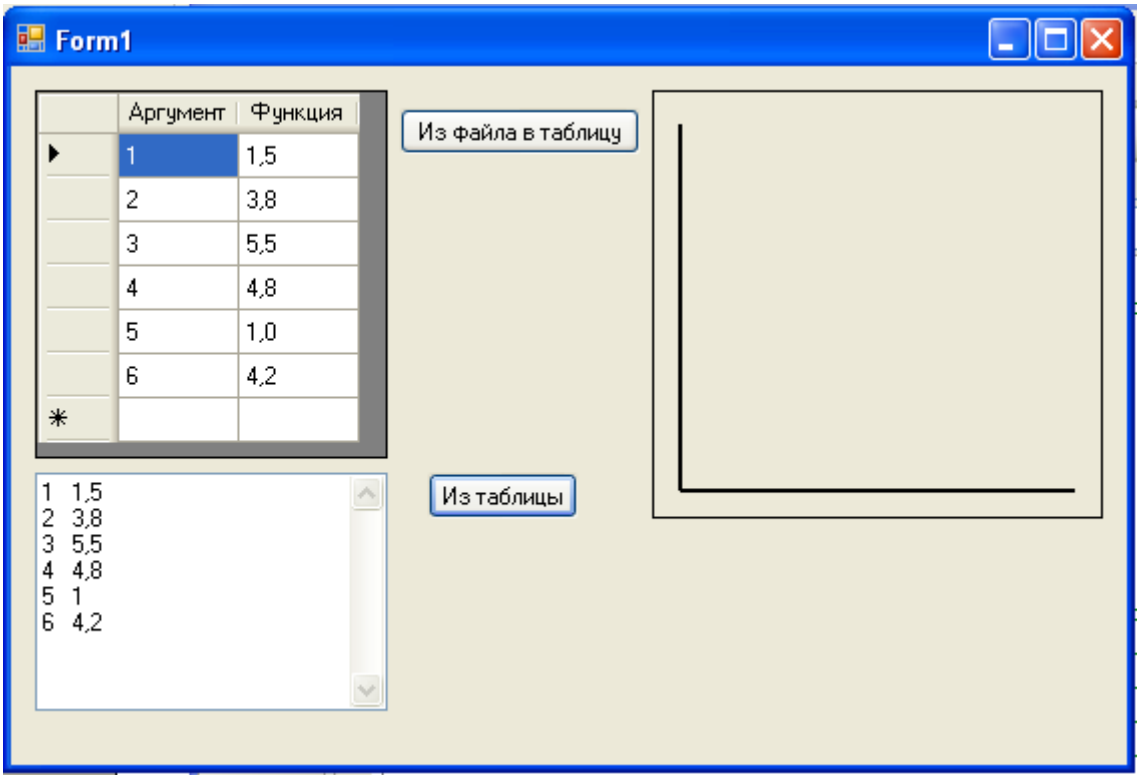
Константа 10 (пикселей) - это отступ от левой и нижней стороны pictureBox для размещения надписей делений по осям. Конечные точки также выбираются с учетом этого отступа.

В общем случае при помощи параллельного переноса осей на величины ox_pix и oy_pix необходимо ввести новую систему координат, у которой ось «y» к тому же должна быть направлена вверх.

Вторая особенность заключается в том, что график строится не в действительных значениях для функции y = f(x), которые расчитает программа, а в пикселях. Поэтому мы должны подобрать масштабы Mx и My для перевода действительных значений переменных в пиксели с целью

увеличения (или уменьшения) размеров графика таким образом, чтобы он был виден на экране. Для переменных кода в пикселях введем далее расширение «pix».

На рисунке – результат работы программы с использованием такого кода.



Правила построения графиков

Чтобы по предлагаемой методике можно было строить графики любых функций, вместо действительных значений предлагается использовать относительные ***безразмерные*** значения переменных.

1. График функции y = f(x) строится в относительных безразмерных величинах (обозначены со штрихом) $x^{|}$, $y^{|}$:

$$x^{|} = x / | x\_max |;$$
$$y^{|} = y / | y\_max |,$$

где | x_max | и | y_max | – наибольшие абсолютные значения переменных x и y. Следовательно, любой график будет находиться внутри прямоугольника

$$-1 =< x^{|} <= 1;$$
$$-1 =< y^{|} <= 1$$

и касаться хотя бы в одной точке двух горизонтальных прямых $y^{|}=1$ и $y^{|}=-1$ и двух вертикальных $x^{|}= 1$ и $x^{|}= -1$.

2. Чтобы получить действительные значения x и y в любой точке графика, необходимо определить относительные значения $x^{|}$ и $y^{|}$ в этой точке и умножить их на наибольшее абсолютное значение | x_max | или | y_max |,

соответственно.

3. Для получения **координатной сетки**, в которой будет находиться график, каждую из двух осей координат x и y можно разбить на некоторое число равных частей, например, на 20 равных частей, и из точек разбиения провести горизонтальные и вертикальные линии.

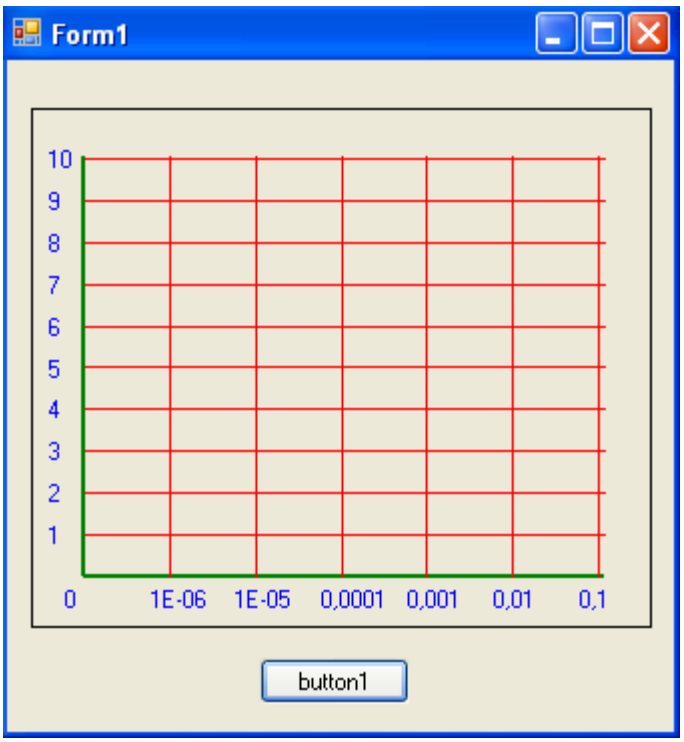
4. Вне координатной сетки следует записать цифровые или буквенные обозначения величин графика.

Далее приведен фрагмент программного кода с комментариями, в котором построена сетка для графика функции, табличные значения которой получены при вычислении суммы бесконечного ряда (варианта задания) .

```
public int NPoints = 6;
int indent_pix = 20;                //Отступ нужен для помещения надписей по осям
public float x_max = 0.1f;          //Значенния x_max и x_min заданы в условии задания
public float x_min = 0.00001f;
int N_step_grid_x = 7;              //Чтобы было 6 вертик. линий в сетке по условию
float step_grid_x;
public int step_grid_x_pix;         //шаг изменения значений аргумента в пикселах
public int step_grid_y_pix;         //шаг изменения значений функции в пикселах
int M_x = 250;
int M_y = 210;
int x_point_end_pix;
int y_max = 5;      //Значения функции y_max и y_min. Для функции примера y_max=5
int y_min = 1;      //y_max и y_min определяются заданием к лабораторной работе

        //Обработчик события Paint для pictureBox1
private void pictureBox1_Paint(object sender, PaintEventArgs e)
{
      //Создаем свой экземпляр класса Graphics
      Graphics myGraphics = e.Graphics;
      //Можно создать экземпляр Graphics иначе (см. следующий оператор)
      //Graphics myGraphics = pictureBox1.CreateGraphics();

      //определяем положение осей координат в пикселях
      int ox_pix = indent_pix;              //координата x начала координат
      int oy_pix = pictureBox1.Height - indent_pix; //коорд. y начала координат
      float x_point_end;                    //Объявление конечной точки по оси x
      x_point_end = 1.0f;                   //Значение конечной точки по оси x по условию

      //Вычисление значения конечной точки оси x в пикселах с использованием
      // масштабного коэффициента и с явным преобразованием типа float в тип int

      x_point_end_pix = (int)x_point_end * M_x - indent_pix;

      //Выбираем зеленое перо толщиной 2;
      Pen greenPen_x = new Pen(Color.Green, 2);
      //Задаем координаты двух граничных точек оси:
      Point point1 = new Point(ox_pix, oy_pix);
```

```
Point point2 = new Point(x_point_end_pix, oy_pix);
//Строим линию через две заданные граничные точки:
myGraphics.DrawLine(greenPen_x, point1, point2);

//Для нашей задачи число горизонтальных линий в сетке равно y_max
int N_step_grid_y = y_max;
//Шаг сетки в направлении оси "y" (высота всей сетки равна 1 единице)
float step_grid_y; //шаг изменения значений функции
int step_grid_y_pix; //шаг изменения значений функции в пикселах
step_grid_y = 1.0f / N_step_grid_y;    //значение шага по оси y
step_grid_y_pix = (int)(step_grid_y * M_y); //шага по оси y в пикселах
//Выбираем красное перо толщиной 1:
Pen redPen = new Pen(Color.Red, 1);

//Строим в цикле горизонтальные линии сетки от нулевой линии (оси x) вверх:
//Для этого определяем координаты граничных точек сетки
int j_y;  //счетчик для цикла
int y1_pix = oy_pix;
for (j_y = 1; j_y <= N_step_grid_y; j_y++)
{
    y1_pix = y1_pix - (int)step_grid_y_pix;
    //Задаем координаты двух граничных точек линии сетки:
    Point point3 = new Point(ox_pix, y1_pix); // левая точка
    Point point4 = new Point(x_point_end_pix, y1_pix); //правая точка
    //Строим прямую линию через две заданные точки:
    myGraphics.DrawLine(redPen, point3, point4);
}
    //Строим ось ординат "oy" от y = 0 до y = 1;
    //Объявляем и задаем ординату последней точки оси ординат "y" при y = 1:
    float y_point_end_pix;
    y_point_end_pix = oy_pix;
    //Выбираем зеленое перо толщиной 2;
    Pen greenPen = new Pen(Color.Green, 2);
    //Задаем координаты двух граничных точек оси y:
    Point point5 = new Point(ox_pix, indent_pix); //Верхн. тчк в pictureBox
    Point point6 = new Point(ox_pix, (int)y_point_end_pix); //нижняя
    //Строим линию через две заданные граничные точки:
    myGraphics.DrawLine(greenPen, point5, point6);

    //Строим вертикальные линии сетки от оси y вправо
    int j_x; float x1;
    int x1_pix;
    step_grid_x = (float)(1.0f / N_step_grid_x); //шаг измен. знач. аргум.
    step_grid_x_pix = (int)(step_grid_x * M_x);  //шаг измен. в пикселах
    x1_pix = ox_pix;
    for (j_x = 1; j_x < N_step_grid_x; j_x++)
    {
        x1_pix = x1_pix + (int)step_grid_x_pix;
        //Задаем координаты двух граничных точек линии сетки:
        Point point7 = new Point(x1_pix, indent_pix);      // левая точка
        Point point8 = new Point(x1_pix, (int)y_point_end_pix);//правая
        //Строим прямую линию через две заданные точки:
```

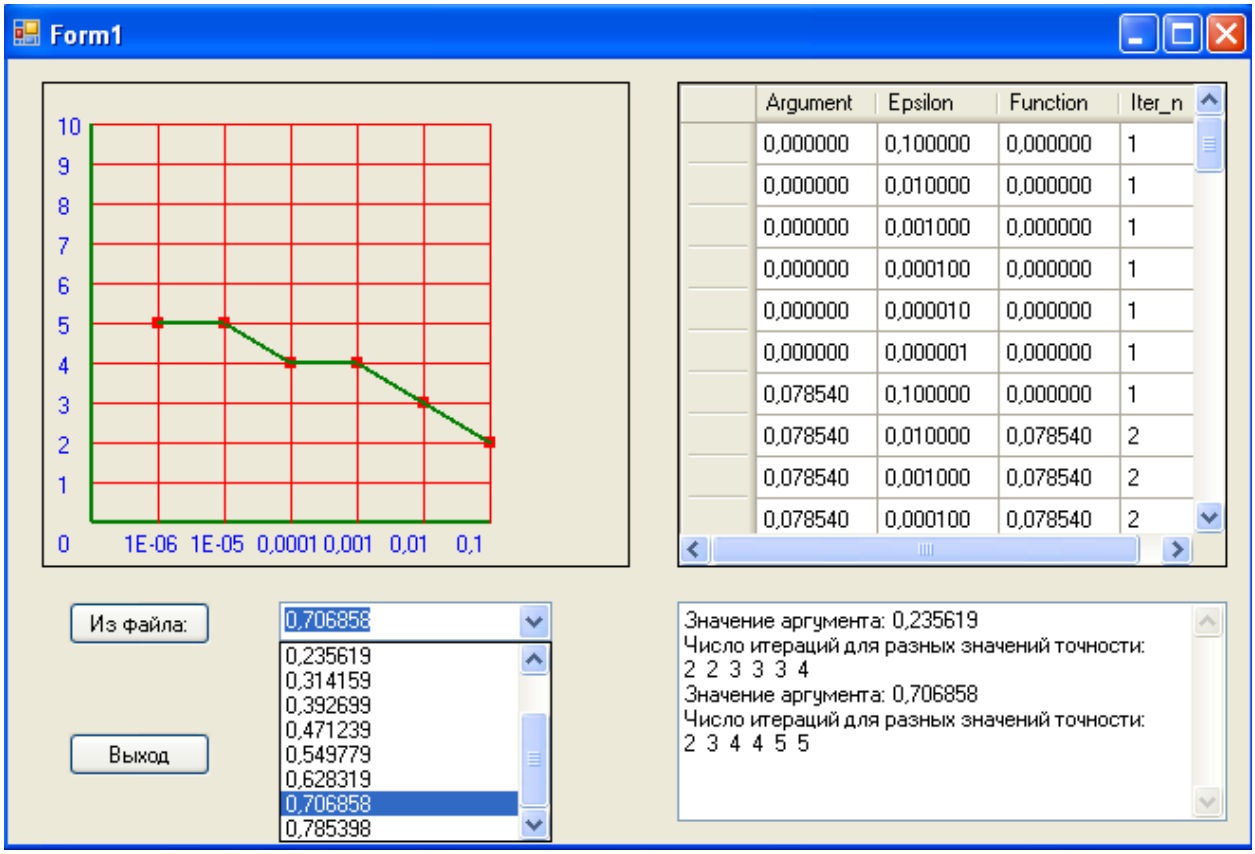
```
            myGraphics.DrawLine(redPen, point7, point8);
        }

    //Записываем числа по осям координат, пользуясь функцией DrawString(msg, . . .) :
    //Объявляем локальные переменные:
int n;  float p1 = 1.0f;  float p2;  string msg;

//Записываем числа по оси "+oy":
for (n = 0; n <= N_step_grid_y - 1; n++)  //9
{
    //p2 = p1 - n * 0.1F; эта формула выводит знач. в виде беск. дроби, поэтому
    // умножаем вычисленное значение на 100.0f, округляем его с помощью
    //функции Math.Round и делим результат на 100.0f
    //p2 = (float)(Math.Round((p1 - n * (float)1 / N_step_grid_y) * 10.0F)) / 10.0F;
    p2 = (float)(Math.Round
            ((p1 - n * (float)1 / N_step_grid_y) * 100.0F)) / 100.0F;
    msg = Math.Round(p2 * N_step_grid_y).ToString();//Получаем значение
    myGraphics.DrawString(msg, this.Font, Brushes.Blue,
                ox_pix - 20, oy_pix - 215 + n * step_grid_y_pix);
}
//Записываем числа по оси "+ox":
float kf = 0.00001f;   //Коэф. для вывода в удобном виде
p2 = 0.0f;             //Нач. значение для вывода нуля в  начале  координат
for (n = 1; n <= 7; n++)
{
    msg = p2.ToString();
    myGraphics.DrawString(msg, this.Font, Brushes.Blue,
           ox_pix - 55 + n * step_grid_x_pix, oy_pix + 5);
    //Чтобы без бесконечной дроби вывелись все значения по оси "ox",
    //умножать и делить приходится на 1000000.0f
    p2 = (float)(Math.Round((0.1F * kf) * 1000000.0f)) / 1000000.0f;
    kf = kf * 10.0f;
}
```




```
//----------------------Теперь строим график---------------------------------
```

Рекомендации к написанию основного кода

На приведенном ниже рисунке приведен пример формы работающего приложения (возможный вариант).



Форма содержит следующие компоненты:

- `TextBox` – для отображения данных, записанных в текстовый файл при выполнении лабораторной работы № 1;
- `Button` – кнопка для чтения данных из файла;
- `DataGridView` – для отображения тех же данных в таблицу, т.е. в виде набора ячеек (строк и столбцов);
- `Panel` – для выделения области, в которую помещается графическое окно с графиками требуемых в задаче зависимостей;
- `PictureBox` – графическое окно;
- `ComboBox` – для выбора значения аргумента.

Рекомендация 1

Данные, записанные в файл при выполнении первой лабораторной работы, содержат 4 элемента (аргумент, точность вычисления, значение функции и число итераций). Их удобно представить в виде типа данных `структура`.

Что такое `структура`? Все типы в C# делятся на две основные разновидности: ***значимые*** и ***ссылочные***. К значимым типам относятся все числовые типы данных (`int`, `double` и др.), а также перечисления (`System.Enum`) и структуры (`struct`). Память для значимых типов выделяется из стека, т.е. из того участка памяти, где располагается и программа, а память для ссылочных типов (классов и интерфейсов) выделяется из «кучи», т.е. той памяти, которой управляет среда исполнения `.NET`.

Доступ к значимым типам производится быстрее, чем к ссылочным типам. При адресации к имени, связанному со значимым типом, производится обращение непосредственно к значению, а при адресации к значению, связанному со ссылочным типом, сначала производится обращение к ссылке, которая указывает на место в памяти, где находится значение.

Структуры очень похожи на классы. Они представляют собой их «облегченный» вариант. Структуры рекомендуется использовать тогда, когда есть всего несколько полей и методов.

Для нашей задачи можно объявить структуру с 4-мя полями:

```
struct myData   //Структура с данными для построения графика
{
  public double argument;    //Значение аргумента
  public double epsilon;     //Значение точности
  public double function;    //Значение функции
  public int n;              //Значение числа итераций
}
```

Массив таких структур будет представлять весь набор данных, хранящихся в файле-результате лабораторной работы № 1. Размер массива определяется условием этой работы.

Рекомендация 2

Чтение данных из файла производится с использованием класса `StreamReader`. Файл с данными рекомендуется поместить в папку проекта. Для удобства выбора файла можно на форму поставить элемент `OpenFileDialog`.

```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
{
    //создаем экземпляр sr класса StreamReader для считывания данных из файла
    System.IO.StreamReader sr = new StreamReader(openFileDialog1.FileName);
    while (sr.EndOfStream != true)   //Пока не достигнут конец потока (файла)
    {
        string valueLine = sr.ReadLine();  //Читаем строку  из файла
            . . .
    }
            . . .
}
```

Для превращения прочитанной строки в массив 4 подстрок, содержащих значения аргумента, точности, функции и числа итераций, можно воспользоваться методом `Split(Convert.ToChar(" "))` для объектов класса `String`. После того, как для каждой строки такой массив создан, можно записать значения в поля структур массива данных. Их можно вывести в окно `textBox`, чтобы убедиться, что считывание прошло верно.

Рекомендация 3

После этого можно приступить к настройке таблицы `DataGridView` и записи значений в ячейки таблицы. На этапе дизайна добавляем в нее нужное число столбцов через свойство Columns (collection) и записываем их названия. В нашем случае число столбцов для записи значений равно 4, а их названия (HeaderText) можно сделать в соответствии со смыслом – *аргумент, точность, функция, итерации*. Имена (Name) столбцов тоже можно сделать осмысленными – argument, epsilon, function, iterations. Нужно помнить, что столбцы в строке нумеруются с 0.

Для добавления строк в таблицу существует метод `Add()`.

А для формирования строки воспользуемся классом `DataGridViewRowCollection`, **связав** его экземпляр с нашей таблицей.

```
DataGridViewRowCollection rows = this.dataGridValue.Rows;
rows.Add(row);            //Добавляем строку  к переменной rows
```

Здесь `row` – массив подстрок для каждой строки, считанной из файла.

Одновременно с добавлением строки в таблицу, можно присваивать значения полям в массиве структур, выбирая эти значения из таблицы.

Рекомендация 4

Для выбора значения аргумента при построении графика удобно поместить на форму элемент ComboBox**.** Поскольку для каждого из 10 значений аргумента в файле содержится по 6 значений числа итераций (для 6 значений точности вычислений), в ComboBox можно поместить только одно значение аргумента (первое или последнее для каждой шестерки). При выборе значения аргумента из ComboBox шесть значений числа итераций для каждого значения точности вычислений могут быть считаны из ячеек DataGridView.

Выделенное значение аргумента может быть зафиксировано в обработчике события comboBox1_SelectedIndexChanged:

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
   int ind = comboBox1.SelectedIndex; //Индекс выделенного значения аргумента
   //Читаем значение числа итераций из поля элемента структуры
   int n_iter = Convert.ToInt32(myArray[comboBox1.SelectedIndex].iter_n);
   flag = true;       //Фиксируем факт выбора аргумента для рисования графика.
   //pictureBox1.Update();     //Метод перерисовывает всю картинку.
   pictureBox1.Invalidate(); //Метод перерисовывает только "инвалидную" часть.
}
```

Рекомендация 5

График, который нужно построить в данной задаче, представляет собой ломаную линию, состоящую из пяти отрезков (для 6 значений точности вычислений). Для рисования каждого отрезка должны быть определены две точки, принадлежащие классу Point. Причем конечная точка каждого отрезка является начальной точкой следующего отрезка графика. Поэтому рекомендуется использовать итеративный алгоритм построения графика. Его структура приведена ниже.

Назовем координдинаты двух точек очередного отрезка графика, например, следующим образом: xx и yy для очередной точки графика и xx_old и yy_old для предыдущей.

если требуется перерисовка графика (выбрано новое значение аргумента)

- задаем координаты xx_old и yy_old начальной точки для рисования графика (будем рисовать график слева направо или справа налево);
- используя массив значений аргумента, находим порядковый номер выделенного в ComboBox значения;
- в цикле от 0 до 5
  - определяем соответствующее значение для числа итераций из таблицы DataGridView;

- определяем значения `xx` и `yy` для точки графика;
- рисуем изображение точки графика на сетке в виде маленького прямоугольника;
- если шаг цикла > 0

  рисуем линию между точкой с координатами `xx_old` и `yy_old` и точкой с координатами `xx` и `yy;`
- делаем точку с координатами `xx` и `yy` предыдущей для следующего отрезка графика: `xx_old = xx; yy_old = yy;`

- конец «цикла»

конец «если»

6. «Последовательности данных»
Часть 1. Динамические массивы

Массивы применяются в программировании для хранения множеств однотипных элементов. В C# для работы с массивами существует класс `System.Array`, предоставляющий набор полезных свойств и методов. Все массивы являются объектами этого класса. Несмотря на то, что массивы – мощный инструмент хранения данных, они имеют несколько ограничений.

1. Нельзя изменить количество элементов в массиве после того, как он создан. Таким образом, массив может оказаться как слишком большим, так и слишком маленьким.
2. Нельзя напрямую вставить элемент в массив, так и удалить его.
3. Доступ к элементам массива осуществляется только при помощи числового индекса.

Для работы с последовательностями данных, размерность которых может меняться в процессе работы программы (с **динамическими** массивами), в C# определен класс **ArrayList**, принадлежащий пространству имен `System.Collections`.

Создание объекта класса `ArrayList` производится обычным для C# способом с использованием оператора `new`:

```
ArrayList myArrayList = new ArrayList();
```

Как видно, при создании не указано число элементов, хранимых в объекте `myArrayList`. По умолчанию, изначально этот объект имеет емкость (свойство `Capacity`), равную 16, что означает, что после создания объекта в нем может храниться не больше 16 элементов. При добавлении элементов емкость будет автоматически увеличиваться.

Для добавления элементов в динамический массив используется метод `Add()`, аргумент которого может иметь любой тип, так как элементы в динамическом массиве хранятся как объекты класса `System.Object` (от этого класса унаследованы все остальные классы C#). При каждом вызове метода `Add()` переданный ему объект добавляется в конец динамического массива. Количество элементов в динамическом массиве можно получить с помощью свойства `Count`. Некоторые свойства и методы класса `ArrayList`, которые можно использовать в программах, представлены в нижеследующих таблицах.

Свойство	Тип	Описание
Capacity	int	Возвращает или устанавливает емкость объекта ArrayList. Емкость – это максимальное количество элементов, которые могут храниться в объекте ArrayList.
Count	int	Возвращает количество элементов в массиве ArrayList.
Item	object	Возвращает или устанавливает значение элемента по указанному индексу. Это свойство является индексатором класса ArrayList.

Метод	Возвращаемый тип	Описание
Repeat() статический	ArrayList	Возвращает новый объект ArrayList, все элементы которого равны переданному значению.
Add()	int	Добавляет новый элемент в конец объекта ArrayList.
AddRange()	void	Добавляет элементы из другой коллекции (например, массива) в конец объекта ArrayList.
BinarySearch()	int	Перегружен. Выполняет двоичный поиск указанного элемента в отсортированном динамическом массиве.
Clear()	void	Удаляет все элементы, содержащиеся в массиве ArrayList.
Clone()	object	Выполняет прямое копирование массива ArrayList. Если элементы массива содержали ссылки на объекты, то будут скопированы ссылки, а не соответствующие им объекты.
Containes()	bool	Определяет, содержится ли указанный элемент в массиве ArrayList.
CopyTo	void	Перегружен. Копирует элементы из массива ArrayList в одномерный массив.
Equals() (Унаследован от Object)	bool	Перегружен. Определяет, равны ли два объекта между собой.
GetEnumerator()	IEnumerator	Перегружен. Возвращает перечислитель, который можно использовать для перебора элементов массива ArrayList.
GetRange()	ArrayList	Возвращает новый массив ArrayList, содержащий диапазон элементов из оригинального массива.
IndexOf()	int	Перегружен. Возвращает номер позиции первого вхождения указанного элемента в массив ArrayList. Номера позиций начинаются с нуля.
Insert()	void	Вставляет элемент на указанную позицию в массиве ArrayList.
InsertRange()	void	Вставляет элементы коллекции на указанную позицию в массиве ArrayList.
LastIndexOf()	int	Перегружен. Возвращает номер позиции последнего вхождения указанного элемента в массив ArrayList. Номера позиций начинаются с нуля.
Remove()	void	Удаляет первое вхождение указанного элемента в массив ArrayList.
RemoveAt()	void	Удаляет элемент на указанной позиции из массива ArrayList.
RemoveRange()	void	Удаляет из массива ArrayList диапазон элементов, начиная и заканчивая указанными позициями.
Reverse()	void	Перегружен. Меняет порядок элементов во всем массиве или в его диапазоне.
SetRange	void	Вставляет элементы коллекции в указанный диапазон массива ArrayList, причем существующие элементы заменяются новыми.

Sort()	void	Перегружен. Сортирует все элементы или диапазон элементов массива ArrayList.
ToArray()	object[]	Копирует элементы массива ArrayList в обычный массив
ToString() (унаследован от Object))	string	Возвращает строку, представляющую текущий объект
TrimToSize()	void	Уменьшает емкость ArrayList до числа элементов, хранящихся в массиве ArrayList в момент вызова метода.

Далее приведен программный код, в котором используются некоторые свойства и методы динамических массивов.

Прежде всего, следует отметить, что системный модуль `System.Collections`, содержащий класс `ArrayList`, по умолчанию не включен в состав модулей проекта. Поэтому его следует подключить с помощью предложения `using System.Collections;`

Программный код имеет большое количество комментариев, которые помогут понять и усвоить особенности применения свойств и методов.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Collections;

namespace WinAppDinEx
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        //Функция для организации вывода динамического массива.
        //Возвращает строку (тип string), состоящую из элементов массива
        //В цикле используется свойство Count(число элементов массива).
        public string ShowArrayList(ArrayList myArrList)
        {
            string value = "";
            for (int counter = 0; counter < myArrList.Count; counter++)
            {
                value = value + myArrList[counter] + " ";
            }
            return value;
        }
```

```
int kol;
private void btnExample_Click(object sender, EventArgs e)
{
    //Создание объекта класса ArrayList
    ArrayList myArrList = new ArrayList();
    //Вывод значения Capacity в textBox
    //tbExample.Text = tbExample.Text + "Значение емкости:  "
    //+ myArrList.Capacity.ToString() + "\r\n";
    string tBvalue;
    //Добавление элементов в динамический массив myArrList
    //с помощью методов Add(), AddRange() и InsertRange().
    myArrList.Add("Для");
    myArrList.Add("добавления");
    myArrList.Add("элементов");
    myArrList.Add("применяется");
    myArrList.Add("метод");
    myArrList.Add("Add()");
    tBvalue = Convert.ToString(ShowArrayList(myArrList));
    tbExample.Text = tbExample.Text + tBvalue + "\r\n";
    string[] myStr1 = { "или", "метод", "AddRange()" };
    myArrList.AddRange(myStr1);
    tBvalue = Convert.ToString(ShowArrayList(myArrList));
    tbExample.Text = tbExample.Text + tBvalue + "\r\n";

    string[] myStr2 = { "или",  "метод", "InsertRange()" };
    myArrList.InsertRange(6, myStr2);
    tBvalue = Convert.ToString(ShowArrayList(myArrList));
    tbExample.Text = tbExample.Text + tBvalue + "\r\n";
    //Вывод значения Capacity в textBox
    tbExample.Text = tbExample.Text + "Значение емкости:  "
                   + myArrList.Capacity.ToString() + "\r\n";
    //Вывод значения Count в textBox
    tbExample.Text = tbExample.Text + "Число элементов:  "
                   + myArrList.Count.ToString() + "\r\n";
    //Уменьшение емкости динамического массива
    myArrList.TrimToSize();
    tbExample.Text = tbExample.Text + "Емкость равна:  "
                   + myArrList.Capacity.ToString() + "\r\n";
    //Удаление одинаковых элементов из динамического массива
    for (int counter = 0; counter < myArrList.Count; counter++)
    {
        if (myArrList[counter].Equals("метод"))
            myArrList.Remove("метод");
    }
    tBvalue = Convert.ToString(ShowArrayList(myArrList));
    tbExample.Text = tbExample.Text + tBvalue + "\r\n";
    //Сортировка динамического массива
    myArrList.Sort();
    tBvalue = Convert.ToString(ShowArrayList(myArrList));
    tbExample.Text = tbExample.Text + tBvalue + "\r\n" + "\r\n";

    //Получение перечислителя при помощи метода GetEnumerator
    //и использование его для чтения элементов массива myArrList
```

```
        IEnumerator myEnumerator = myArrList.GetEnumerator();
        tbExample.Text = tbExample.Text +
                                "Просмотр с myEnumerator: " + "\r\n";
        while (myEnumerator.MoveNext())
        {
             tbExample.Text = tbExample.Text +
                                    myEnumerator.Current + "  ";
        }
        tbExample.Text = tbExample.Text + "\r\n" + "\r\n";
       //Использование цикла foreach для чтения
       //элементов массива myArrList
       tbExample.Text = tbExample.Text +
                         "Просмотр с циклом foreach: " + "\r\n";
        foreach (string myString in myArrList)
        {
            tbExample.Text = tbExample.Text + myString + " ";
        }
        tbExample.Text = tbExample.Text + "\r\n" + "\r\n";
    }
```

Рекомендации к выполнению задания 2
(в лабораторной работе «Последовательности данных»)

Оформить один из алгоритмов сортировки для динамического массива.

1. Создайте форму для визуального оформления проекта.
2. Предусмотрите на ней два окна textBox для вывода значений числового динамического массива и отсортированного массива.
3. Для выбора количества элементов массива используйте элемент comboBox. Воспользуйтесь его свойствами `Items` и `SelectedIndex`.
4. Случайные значения элементов динамического массива получите с помощью класса `Random` и его метода `Next`.
5. Воспользуйтесь методом `Clone()` для получения второго объекта `ArrayList` – динамического массива `myAlClone` (имя взято для примера), который будет использоваться для написания одного из алгоритмов сортировки. Ниже приведен пример кода с использованием метода `Clone()`.

```
ArrayList myAL = new ArrayList();
ArrayList myAlClone = new ArrayList();
myAlClone = (System.Collections.ArrayList)myAL.Clone();
```

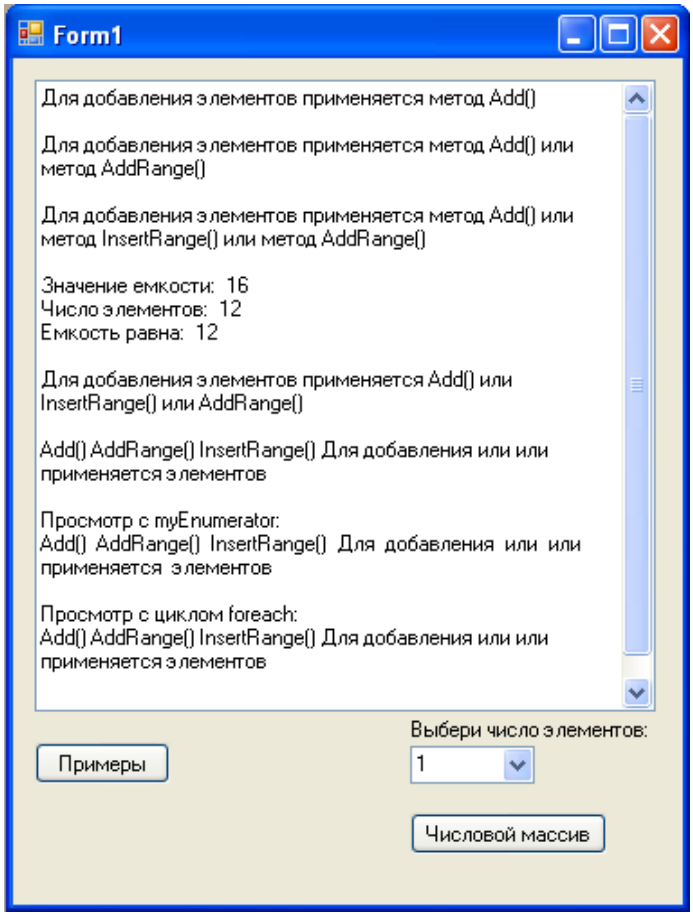
6. Напишите для динамического массива `myAlClone` один из алгоритмов сортировки, исследованных в лабораторной работе «Алгоритмы сортировки».
7. Проведите проверку правильности сортировки с помощью метода, написанного в лабораторной работе «Алгоритмы

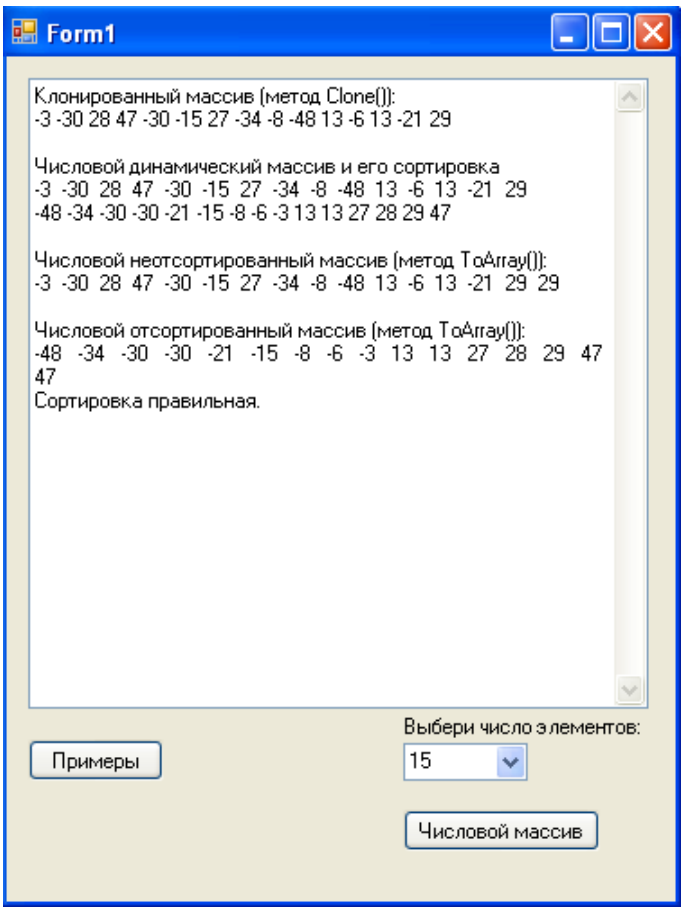
сортировки». Поскольку этот метод был написан для целочисленных входных массивов (тип `int`), в данном случае можно выбрать один из следующих способов:

- переписать алгоритм для случая, когда входными массивами являются динамические массивы,
- воспользоваться уже написанным методом, предварительно переведя динамические массивы в обычный целочисленный тип.

Для выполнения указанного перевода нужно воспользоваться методом `ToArray`.

На рисунках риже приведены образцы форм работающего проекта.




Form1
Клонированный массив (метод Clone()):
-3 -30 28 47 -30 -15 27 -34 -8 -48 13 -6 13 -21 29
Числовой динамический массив и его сортировка
-3 -30 28 47 -30 -15 27 -34 -8 -48 13 -6 13 -21 29
-48 -34 -30 -30 -21 -15 -8 -6 -3 13 13 27 28 29 47
Числовой неотсортированный массив (метод ToArray()):
-3 -30 28 47 -30 -15 27 -34 -8 -48 13 -6 13 -21 29 29
Числовой отсортированный массив (метод ToArray()):
-48 -34 -30 -30 -21 -15 -8 -6 -3 13 13 27 28 29 47 47
Сортировка правильная.
Выбери число элементов:
15
Примеры
Числовой массив

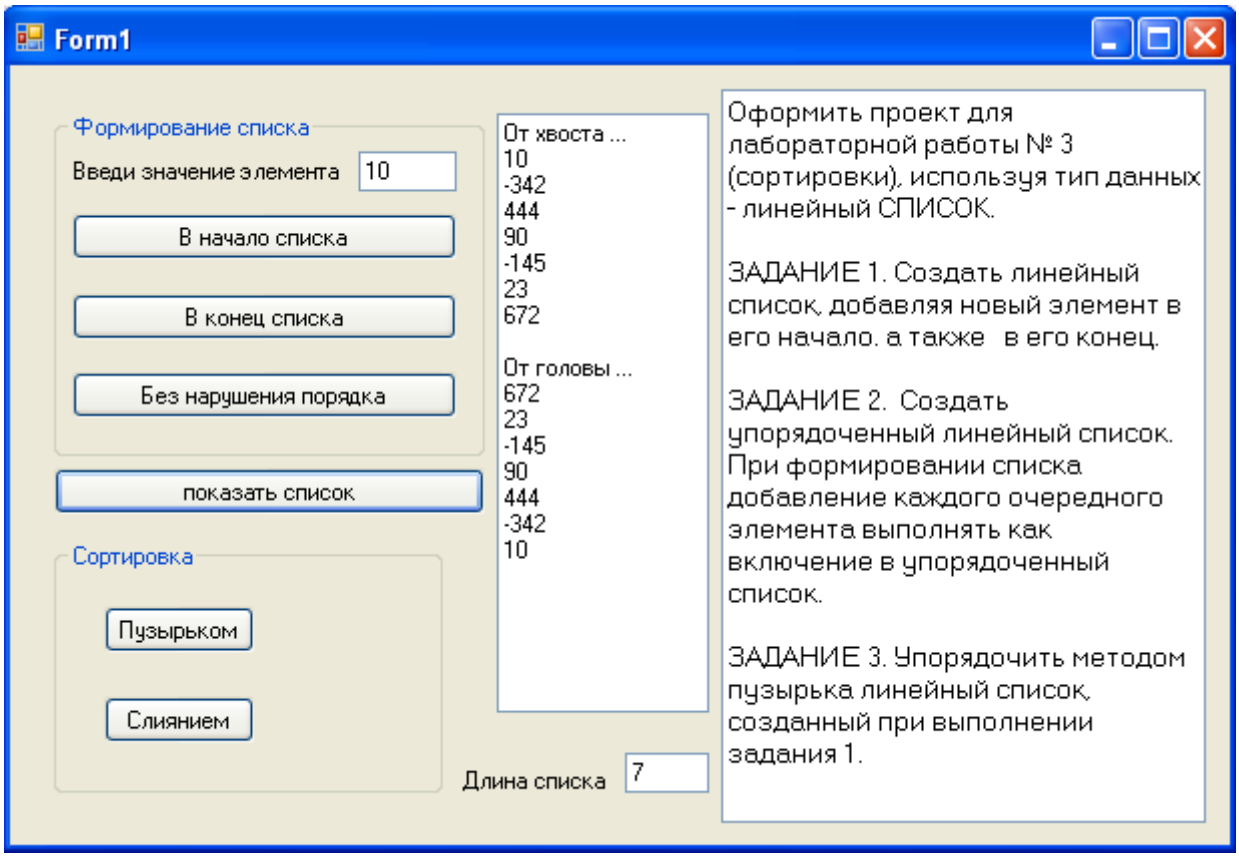
«Последовательности данных»
Часть 2. Связанные списки

При создании приложений, в которых требуется размещать в памяти последовательности данных одного типа, иногда бывает удобно представлять эти данные в виде **связанных списков** элементов, в которых каждый элемент указывает на хронологически связанный с ним элемент. Поэтому каждый элемент списка должен быть объектом, содержащим в себе информацию об этой связи.

Применение типа Pointer для создания таких списков, позволяющего осуществлять непосредственный доступ и внесение изменений в системную память, в C# может быть осуществлено только с помощью кода, помеченного как **unsafe (небезопасный).** Использование этого режима представляет потенциальную **опасность,** поскольку в данном случае информация может записываться в критические системные участки памяти с возможными катастрофическими результатами. Поэтому в данной лабораторной работе рекомендуется **не использовать Pointer**.

В языке C# имеется специальный класс для представления указанной структуры данных. Однако, в лабораторной работе требуется создать собственный класс и реализовать ряд алгоритмов со списками на его основе.

Ниже приведен возможный вид формы работающего приложения с текстом задания.



Образец объявления класса и его членов для нашей задачи

```
public class ListNumber
{
    public int value;         //поле для значения целого числа
    public ListNumber next;   //поле для связи со следующим элементом
    public ListNumber prev;   //поле для связи с предшеств. элементом
    public ListNumber head;   //Голова списка
    public ListNumber tail;   //Хвост списка
};
```

Что интересно в этом объявлении? Оказывается, члены класса могут объявляться как объекты, принадлежащие к этому же классу. Таким образом, нам удается объявить структуру, содержащую значение целого числа, как информационную часть, и ряд «указателей» на нужные нам элементы списка, объявив соответствующее число объектов:
- next (для указания на следующий элемент списка)
- prev (для указания на предшествующий элемент списка)
- head (для указания на начало списка)
- tail (для указания на конец списка).

Все дело в том, что C# автоматически управляет распределением памяти для создаваемых объектов и сам связывает их с соответствующими указателями, освобождая программиста от этой работы. Объекты – это ссылочные типы. Когда мы присваиваем объект некоторой переменной, мы на самом деле присваиваем этой переменной указатель, который ссылается на этот объект. Создание объекта – состояние, когда происходит первоначальное создание экземпляра объекта. Такая инициализация известна под названием **создание объекта** и осуществляется **конструктором** объекта.

В приведенном ниже фрагменте программного кода (обработчик события нажатия на кнопку «в начало списка») объекты создаются с помощью метода **new** – конструктора по умолчанию. Приведенные комментарии объясняют назначение отдельных операторов программного кода.

```
//Создаем вспомогательный экземпляр объекта для хранения промежуточных
//величин. Он понадобится при работе алгоритмов со списками.
public ListNumber dop = new ListNumber();

public int quantity = 0; //Объявляем переменную для числа элементов в списке


 //Создание списка с добавлением элемента в начало (голову) списка
 private void btnBegin_Click(object sender, EventArgs e)
 {
     if (quantity == 0)
     {
//Если в списке нет ни одного элемента, создаем первый экземпляр с данными

     ListNumber num = new ListNumber();
     //Он и является указателем адреса объекта
     num.value = Convert.ToInt32(textBoxInput.Text);  //вводим значение поля
                                                      //с числом
```

```
//Элементы в список добавляем в начало списка
num.next = null;  //значение поля next для первого созданного элемента
num.prev = null;  //значение поля prev для первого созданного элемента
num.head = num;   //Головой списка становится первый введенный элемент
dop.head = num.head;    //запоминаем голову (начало) списка
dop.tail = num;           //запоминаем хвост списка
quantity = quantity + 1;
}
else
  {
     ListNumber num = new ListNumber();

     num.value = Convert.ToInt32(textBox3.Text);

     num.head = num; //Головой списка становится ссылка на очередной
                     // введенный элемент
     num.next = dop.head; //в поле next заносим ссылку на ранее
                          //введенный элемент
     num.next.prev = num; //в поле prev ранее введенного элемента
                     //заносим ссылку на очередной введенный элемент
     dop.head = num.head; //запоминаем голову (начало) списка

     quantity = quantity + 1;

  }
}
```

Для образца приведен программный код для вывода элементов списка «с головы до хвоста».

```
//вывод элементов списка с головы до хвоста - от последнего созданного
//элемента до первого
  private void btnShow_Click(object sender, EventArgs e)
  {
      //создание объекта для перемещения по списку
      ListNumber current = new ListNumber();
      current.next =dop.head;   //текущий указатель
      string sOut = "";

      textBoxShow.Text = textBoxShow.Text + "От хвоста ..." + "\r\n";

      //Цикл формирования строки для вывода значений из списка
      while (current.next != null)
      {
          sOut = sOut + current.next.value + "\r\n";
          current.next = current.next.next; //изменение текущей ссылки
      }
      //вывод строки в textBoxShow
      textBoxShow.Text = textBoxShow.Text+ sOut + "\r\n";

      textBoxShow.Text = textBoxShow.Text + "От головы ..." + "\r\n";
      . . .
  }
```

Используя приведенные фрагменты кода для обучения, выполните все задания лабораторной работы.

7. Комбинаторные алгоритмы

ОПРЕДЕЛЕНИЯ

1. В математике есть понятие СОЕДИНЕНИЯ.
   Они возможны трех типов.
   - РАЗМЕЩЕНИЯМИ из n элементов по m называются такие их соединения, которые различаются друг от друга самими элементами или их порядком.

     Например, Размещениями из трех элементов a,b,c по 2 будут соединения ab,ac,ba,bc,ca,cb.

     Число размещений из n по m вычисляется по формуле n!/(n-m)!.
   - ПЕРЕСТАНОВКАМИ из n элементов называются их соединения, отличающиеся друг от друга только порядком входящих в них элементов. Для тех же элементов это будут соединения abc, acb, bac, bca, cab, cba. Число перестановок из n элементов равно n!.
   - СОЧЕТАНИЯМИ из n элементов по m называются такие их соединения, которые различаются друг от друга только самими элементами.

     Для тех же элементов это соединения ab, ac, bc. Число сочетаний из n элементов по m равно n!/((n-m)! / m!).

Условие лабораторной работы состоит из трех заданий:

1. Задана последовательность из *n* различных русских букв. Создать Windows-приложение для генерации всех размещений этих букв по *m* элементов, для всех возможных значений *m*.

Входные величины: последовательность букв читается из текстового файла.

Результаты выводятся в текстовое окно на экран и в текстовый файл в виде последовательности пронумерованных строк, в каждой строке – одно размещение.

2. Задана последовательность из *K* различных слов. Написать программу генерации всех перестановок этих слов.

Входные величины: последовательность слов читается из текстового файла.

Результаты выводятся в текстовое окно на экран и в текстовый файл в виде последовательности пронумерованных строк, в каждой строке – одна перестановка.

3. Задана последовательность из 7 кодов цветов, пронумерованных от 1 до 7. Написать программу генерации сочетаний в наборе из *L* цветов (1<=*L*<=7) по *P* элементов для всех возможных значений *P*. Набор создается из первых *L* цветов последовательности.

Входные величины: число цветов *L* можно выбрать из 7-ми разных значений, заданных на форме в ComboBox.

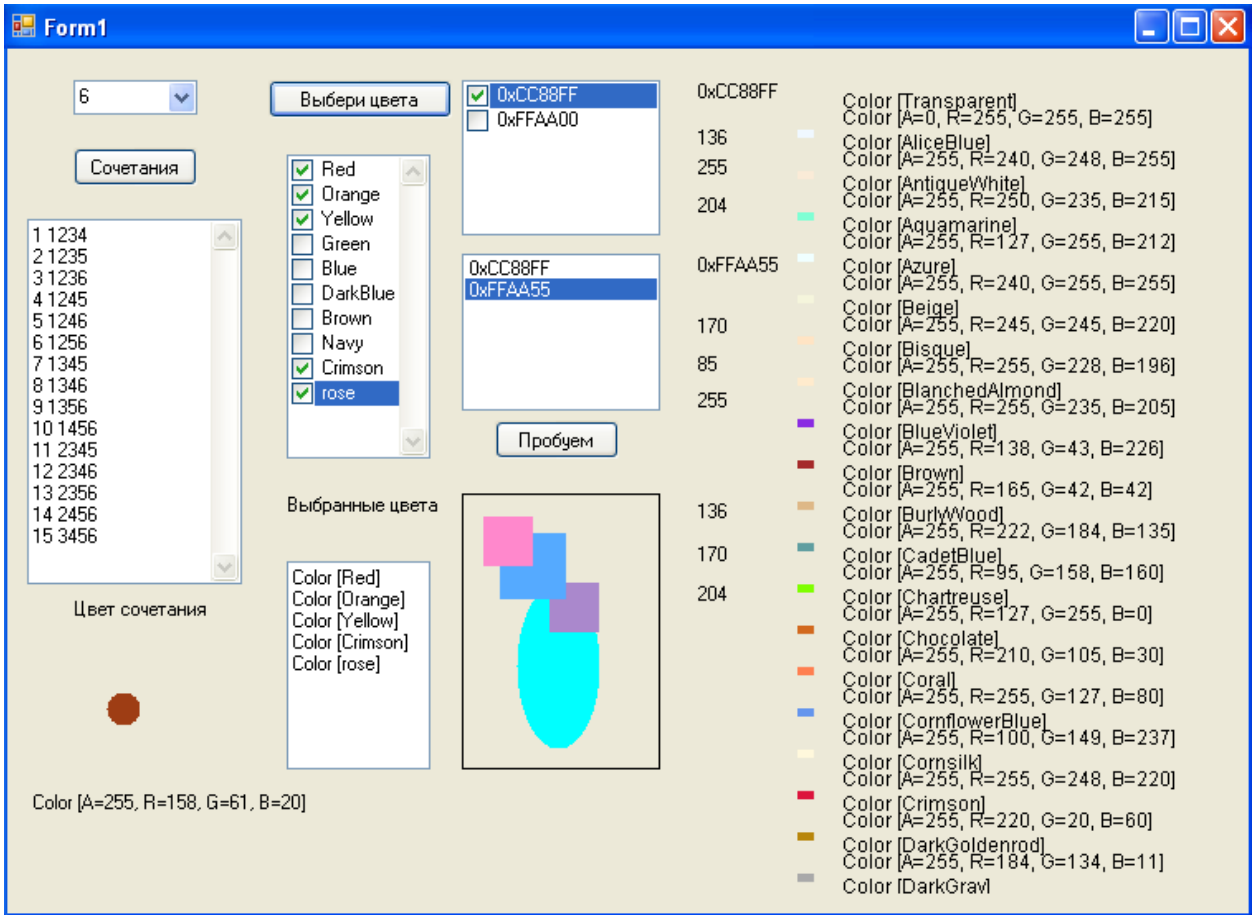
Результаты выводятся в графическое окно в виде последовательности простейших изображений – цветовых пятен любого вида (квадраты, круги, с контуром или без него). Каждое сочетание отображается отдельным пятном, цвет которого формируется на основе пропорционального вклада цвета каждого элемента сочетания.

Для выполнения третьего задания (получения цвета сочетания) необходимо познакомиться с ответствующими объектами, применяемыми в C# для работы со значениями цвета, со способами представления различных цветов и с методами, которыми можно пользоваться**.**

Поэтому для начала мы рекомендуем выполнить дополнительную работу – создать приложение, форма которого приведена ниже.

Работу над этим приложением можно разбить на две части:

1. Изучить и опробовать способы задания цвета в C#, соответствующие классы, структуры и методы. Компоненты для реализации этой части приложения расположены на рисунке справа: графическое окно для вывода текстовой и графической информации о цветах; управляющие элементы для выбора необходимого цвета (ListBox и CheckedListBox); графическое окно и метки для демонстрации результата работы приложения, необходимые кнопки.
2. Научиться работать с одним сочетанием – определять его цвет и выводить нужное в этом случае графическое изображение (цветовое пятно).



Все цвета, зарегистрированные в системе (**system-defined)**, задаются с помощью перечисления **KnownColor enumeration.** Эти цвета можно получить следующим образом:

```
String[] nm = Enum.GetNames(typeof (System.Drawing.KnownColor));
```

Работа с цветом в C# обеспечивается **структурой Color.**

Поля структуры содержат **имена system-defined** цветов, заданных в формате **RGB (**например, Red, SeaGreen, Silver и т.д.**)**. Кроме этого, для нашей задачи важны поля с именами **R, G, B**, содержащие значения красной, зеленой и голубой составляющих цвета структуры. Откройте окно справки и просмотрите набор полей структуры.

Методы определяют способы работы с соответствующими переменными. Ниже приведены лишь некоторые из них, а именно те, которые будут использоваться при написании кода лабораторной работы.

Имя метода	Описание
FromArgb	Creates a **Color** structure from the four 8-bit ARGB components (alpha, red, green, and blue) values.
FromKnownColor	Creates a **Color** structure from the specified predefined color.
FromName	Creates a **Color** structure from the specified name of a predefined color.
ToArgb	Gets the 32-bit ARGB value of this **Color** structure.
ToKnownColor	Gets the KnownColor value of this **Color** structure.
ToString	Converts this **Color** structure to a human-readable string.
GetBrightnes	Gets the hue-saturation-brightness (HSB) brightness value for this **Color** structure.

В приведенном ниже фрагменте кода для создания необходимого массива структур цветов используется метод **FromKnownColor;**

```
Graphics gr = pictureBox1.CreateGraphics();

Color someColor = Color.FromArgb(0); // Структура для временного хранения
Color[] arr = new Color[7];          // Массив структур цветов

someColor = Color.FromKnownColor(KnownColor.Green);
arr[0] = someColor;
someColor = Color.FromKnownColor(KnownColor.Orange);
arr[1] = someColor;
someColor = Color.FromKnownColor(KnownColor.SeaGreen);
arr[2] = someColor;
someColor = Color.FromKnownColor(KnownColor.Red);
arr[3] = someColor;
someColor = Color.FromKnownColor(KnownColor.DarkBlue);
arr[4] = someColor;
someColor = Color.FromKnownColor(KnownColor.Magenta);
arr[5] = someColor;
someColor = Color.FromKnownColor(KnownColor.ForestGreen);
arr[6] = someColor;
```

Далее представлен программный код с комментариями для выполнения дополнительного задания к лабораторной работе № 7

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace ProjComb
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        int kol;                //Число элементов для получения сочетаний (n)
        int m = 4;                          //Сколько элементов в сочетании
        int[] arrComb = new int[50];        //Массив для хранения сочетаний
        Color[] masColor = new Color[7]; //Массив цветов для получения цвета
                                                    //сочетания
        Color colComb;                      //Переменная для цвета сочетания
        int orderComb = 0;                  //переменная для порядкового номера
                                                    //сочетания
        int[] combArr = new int[4];         //Вспомогательный массив
        string value = "";                  //Строка для вывода значений


        //рекурсивная процедура генерирования сочетаний из kol элементов по m
        private void comb(int k)
        {
            for (int i = arrComb[k - 1] + 1; i <= kol - m + k; i++)
            {
                arrComb[k] = i;
                if (k == m)
                {
                    orderComb = orderComb + 1;    //Порядковый номер сочетания
                    for (int j = 1; j <= m; j++)
                    {
                        value = value + Convert.ToString(arrComb[j]);
                    }
                    //вывод сочетания в textBox1
                    textBox1.Text = textBox1.Text + orderComb.ToString() +
                                    " " + value + "\r\n";
                }
                else
                {
                    comb(k + 1);
                };
                value = "";
            }
        }

        //Вызов прцедуры генерирования сочетаний по нажатию кн. "Сочетания"
        private void btnComb_Click(object sender, EventArgs e)
        {
            textBox1.Text = "";
            arrComb[0] = 0;
            comb(1);
        }
```

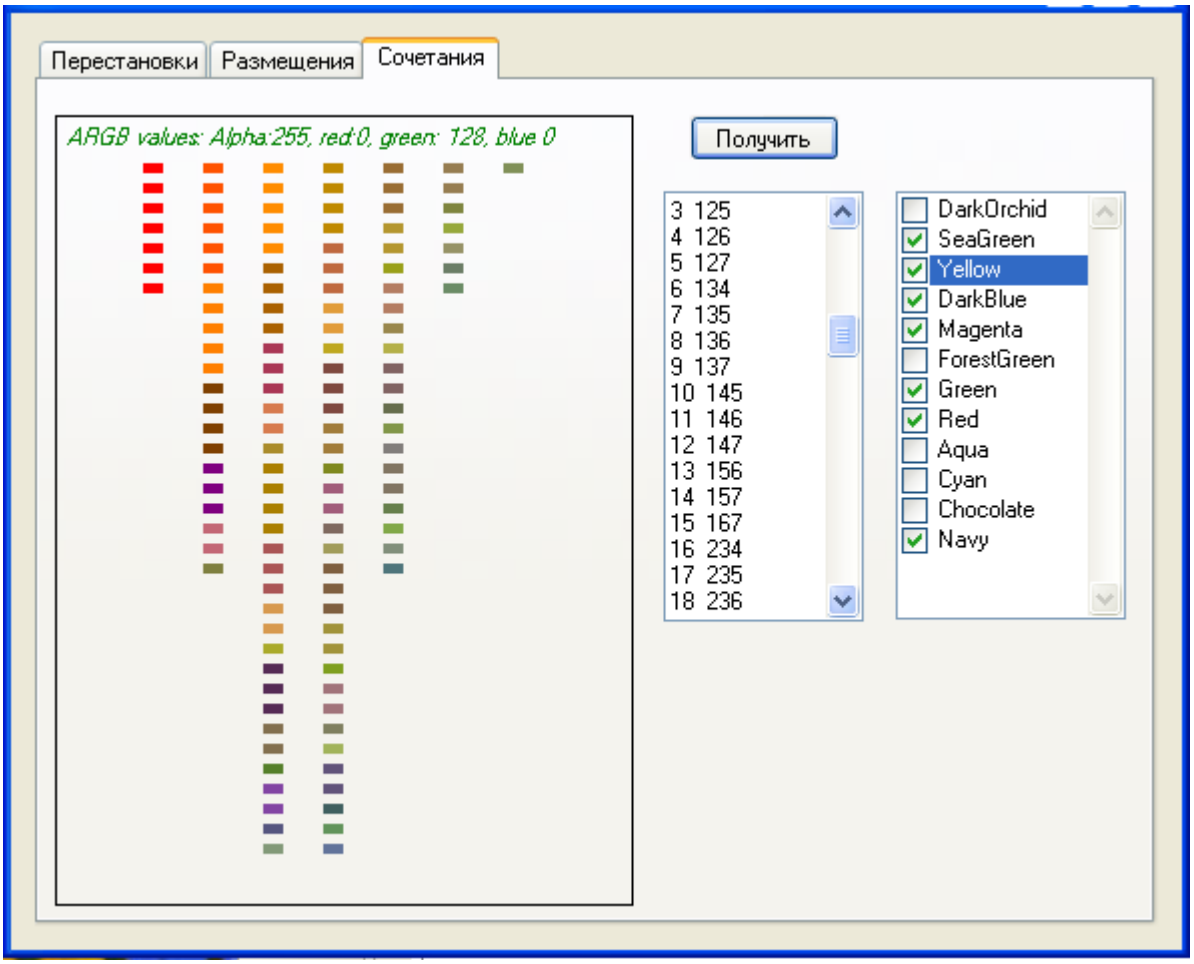
```
    //Выбор цветов из checkedListBox2 и запись их в массив masColor
    private void button1_Click(object sender, EventArgs e)
    {
        int itemCount = checkedListBox2.CheckedItems.Count;
        if (itemCount > 0)
        {
            for (int i = 0; i < itemCount; i++)
            {
                masColor[i] = Color.FromName
                            (checkedListBox2.CheckedItems[i].ToString());
                textBox2.Text = textBox2.Text + masColor[i].ToString()
                                                                + "\r\n";
            }
        }
        else
        {
            MessageBox.Show("Выбери цвета!");
        }

        combArr[0] = 1; combArr[1] = 3; combArr[2] = 4; combArr[3] = 5;
        CombColor();
        label8.Text = colComb.ToString();
        Graphics pbox3 = pictureBox3.CreateGraphics();
        SolidBrush myBr = new SolidBrush(colComb);
        pbox3.FillEllipse(myBr, 20, 20, 20, 20);
    }

    //Определение цвета конкретного сочетания
    private void CombColor()
    {
        int r = 0; int g = 0; int b = 0;
        //Получение суммарных значений R-G-B составляющих цвета сочетания
        for (int j = 0; j < m; j++)
        {
            r = r + masColor[combArr[j]].R;
            g = g + masColor[combArr[j]].G;
            b = b + masColor[combArr[j]].B;
        }
        r = r / m; g = g / m; b = b / m;    //усреднение R-G-B значений
        colComb = Color.FromArgb(r, g, b); //Получение цвета сочетания
    }

private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
    {
        if (comboBox1.SelectedItem != null)
        {
            kol = Convert.ToInt32 (comboBox1.SelectedItem);
            textBox1.Text = "Число цветов:" + " " + kol.ToString() +
                                                                "\r\n";
        }
    }

    //Следующая процедура предназначена для вывода в окно pictureBox2
    //графической информации о некоторых цветах, определенных в
    //перечислении KnownColor: их названии, R-G-B составляющих цвета
    //и графического изображения - прямоугольника соответствующего цвета
    //Напишите коментарии ко всем операторам этой процедуры!
    private void pictureBox2_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;  Color someColor = Color.FromArgb(0);
        KnownColor[] colorMatches = new KnownColor [100];     //[167];
        int count = 0;
        for (KnownColor enumValue = 0; enumValue <= KnownColor.Red;
                                                        enumValue++)
```

```
            {
                someColor = Color.FromKnownColor(enumValue);
        if (someColor.G != 0 && someColor.R != 0 && !someColor.IsSystemColor)
                    colorMatches[count++] = enumValue;
            }
            SolidBrush myBrush1 = new SolidBrush(someColor);
            Font myFont = new Font("Arial", 9);  int x = 5; int y = 5;
            for (int i = 0; i < count; i++)
            {
                someColor = Color.FromKnownColor(colorMatches[i]);
                myBrush1.Color = someColor;
                g.FillRectangle(myBrush1, x, y, 10, 5);
         g.DrawString(someColor.ToString(), myFont, Brushes.Black, x + 25, y);
                someColor = Color.FromArgb(someColor.ToArgb());
    g.DrawString(someColor.ToString(), myFont, Brushes.Black, x + 25, y + 10);
                y += 25;
            }
        }

        //Процедура для работы с 16-ричными кодами цветов
        //код задается 16-ричным значением, или читается из checkedListBox
        //или ListBox.
        private void btnTest_Click(object sender, EventArgs e)
        {
            Graphics pbox = pictureBox1.CreateGraphics();
            Pen pboxPen = new Pen(Color.Black, 3);
            pbox.FillEllipse(Brushes.Cyan, 30, 50, 50, 100);
            int winColor = 0xCC88AA;  Задаем цвет 16-ричным кодом
            //Создаем экземпляр winCol структуры Color с помощью класса
            //ColorTranslator и его метода FromWin32(winColor)
            Color winCol = ColorTranslator.FromWin32(winColor);
            //Вычисляем R-G-B составляющие цвета, выводим значения в метки
            byte g1 = winCol.G;   label4.Text = g1.ToString();
            byte r1 = winCol.R;   label5.Text = r1.ToString();
            byte b1 = winCol.B;   label6.Text = b1.ToString();
            //Создаем экземпляр Color с помощью метода FromArgb(r, g, b)
            Color myNewColor = Color.FromArgb(r1, g1, b1);
            pbox.FillRectangle(new SolidBrush(myNewColor), 50, 50, 30, 30);

            //используем выбор 16-ричных кодов из listBox1 и checkedListBox1
            if ((listBox1.SelectedItem != null)||
                                    (checkedListBox1.SelectedItem != null))
            {
                //Читаем строку (16-ричную константу) из listBox1
                string colMy = listBox1.SelectedItem.ToString();
                label7.Text = colMy;    //выводим строку-значение в метку

                //Преобразуем строку в int-значение
               int winCol1 = Convert.ToInt32 (colMy, 16); //Обратите внимание
               //Создаем экземпляр myWinCol1 структуры Color
                Color myWinCol1 = ColorTranslator.FromWin32(winCol1);
                //Вычисляем R-G-B составляющие цвета, выводим в метки
                byte g = myWinCol1.G;   label1.Text = g.ToString();
                byte r = myWinCol1.R;   label2.Text = r.ToString();
                byte b = myWinCol1.B;   label3.Text = b.ToString();
                //Создаем экземпляр структуры Color с помощью метода
                //FromArgb(r, g, b)
                Color myNewColor1 = Color.FromArgb(r, g, b);
                //Рисуем прямоугольник созданным цветом
             pbox.FillRectangle(new SolidBrush(myNewColor1), 20, 20, 40, 40);

                //Читаем строку (16-ричную константу) из checkedListBox1
                string colMyCb = checkedListBox1.SelectedItem.ToString();
                label11.Text = colMyCb;
```

```
                int winColCb = Convert.ToInt32(colMyCb, 16);
                Color myWinColCb = ColorTranslator.FromWin32(winColCb);
                byte g2 = myWinColCb.G; label12.Text = g2.ToString();
                byte r2 = myWinColCb.R; label13.Text = r2.ToString();
                byte b2 = myWinColCb.B; label14.Text = b2.ToString();
                Color myNewColor2 = Color.FromArgb(r2, g2, b2);
             pbox.FillRectangle(new SolidBrush(myNewColor2), 10, 10, 30, 30);
            }
        }
    }
}
```

Теперь можно приступать к выполнению 3-го задания.

На рисунке ниже приведен вариант формы для выполнения заданий лабораторной работы № 7 (работающий вариант – открыта закладка «Сочетания»).



Размещения и перестановки также можно представить графически, используя отличный от сочетаний способ.

Для работы с файлами при выполнении лабораторной работы рекомендуется использовать компоненты OpenFileDialog и SaveFileDialog.

На форме должен обязательно быть представлен текст заданий лабораторной работы.

8. Алгоритмы обработки символьных строк

Лабораторная работа предполагает создание проекта с несколькими формами (по крайней мере, с двумя).

Задание 1

В файле содержится текст на русском языке, слова записаны строчными буквами без переносов. Написать программу, определяющую, сколько раз встречается в этом тексте набор символов, заданный в символьной строке-образце. Поиск **требуется** организовать в **массиве символов**, в который переписывается весь текст целиком или последовательно отдельные фрагменты текста (см. алгоритм Ю.Л. Костюка). По желанию разработчика можно **дополнительно** реализовать поиск непосредственно в содержимом текстового окна.

Задание 1 может быть реализовано в проекте, состоящем из одной формы. Для выделения цветом найденной подстроки в тексте, необходимо использовать компонент *richTextBox*. Для выбора цвета при отображении найденной подстроки в *richTextBox* потребуется использование структуры *Color*, и знакомство с необходимыми свойствами и методами окна *richTextBox – SelectionColor, Select(,)* и другими.

Для чтения данных из файла необходимо применить компонент *openFileDialog*. Ниже приведен фрагмент кода для организации чтения данных из текстового файла с использованием этого компонента.

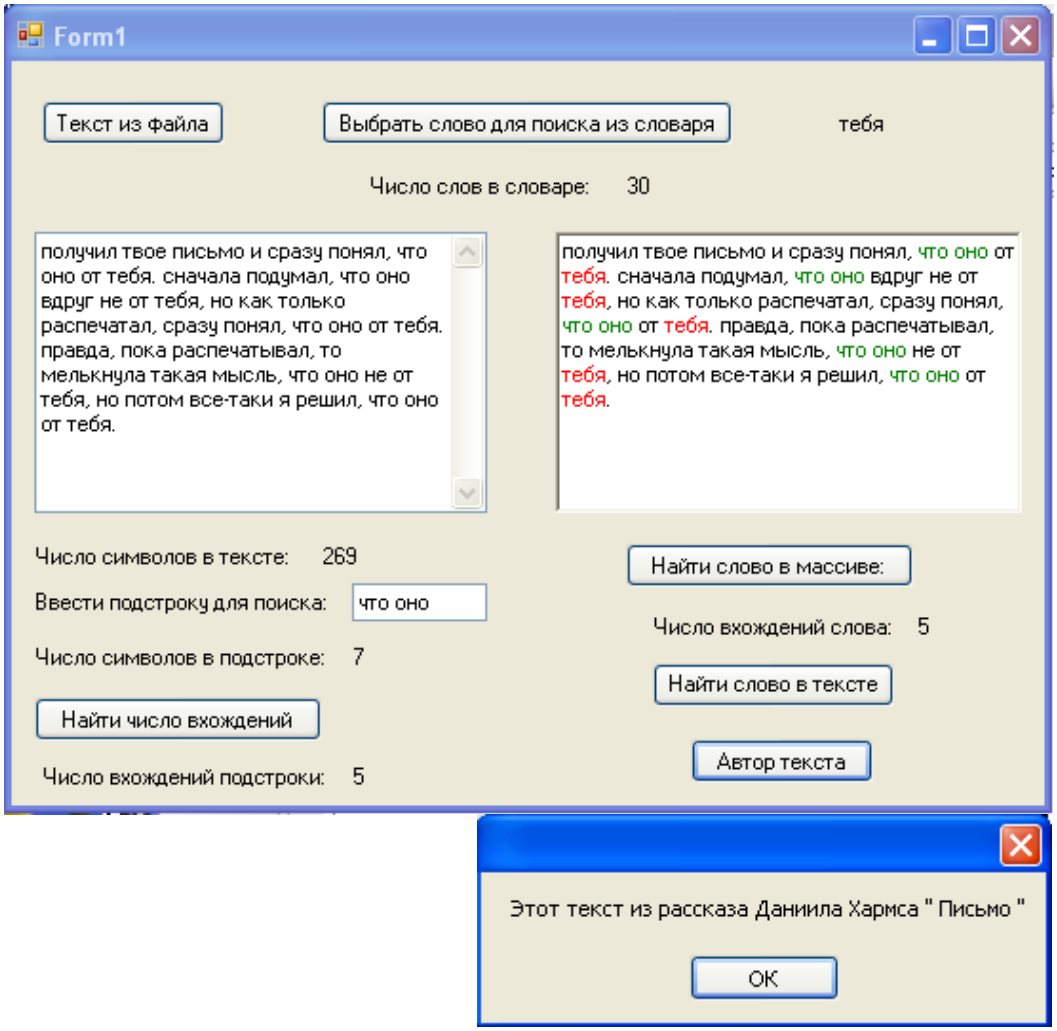
```
if (openFileDialog1.ShowDialog()== DialogResult.OK)
{
    textBoxInput.Text = "";
    string strLine = "";
    System.IO.StreamReader sr = new
    System.IO.StreamReader(openFileDialog1.FileName,
                           Encoding.GetEncoding(1251));
    strLine = sr.ReadLine();
        . . .
```

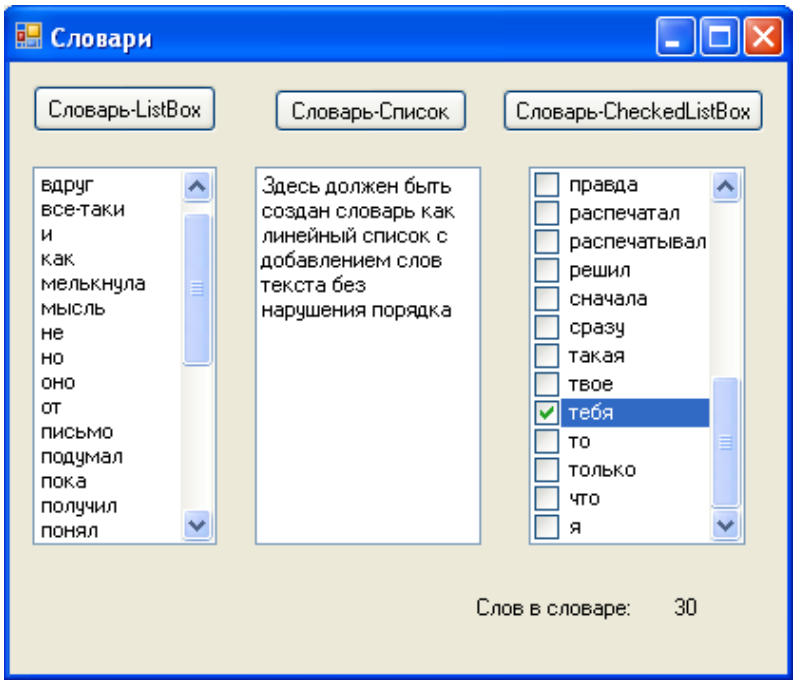
Если файл создан в блокноте или в WordPad, его надо сохранить с расширением *txt*, и указать параметр, равным *Encoding.GetEncoding(1251)*.

Задание 2

Написать программу, создающую два (или три) одинаковых словаря всех слов текста, используемого при выполнении задания 1. Слова в словаре должны располагаться в лексикографическом порядке. Для создания одного из словарей можно использовать компоненты *ListBox* или *CheckedListBox*. Должен быть также создан словарь – линейный список с добавлением в него новых слов без нарушения лексикографического порядка. При выполнении задания 2 в проект требуется добавить вторую форму, на которой организовать создание словарей.

Образцы форм в работающем состоянии





Как создать приложение с двумя формами

Основным строительным блоком при создании Windows-приложений является *форма*. Обычное Windows-приложение обязательно содержит одну главную форму, которая открывается при вызове метода *Run* в процедуре *Main.*

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new Form1());
}
```

Но приложение может содержать несколько *форм,* которые по желанию пользователя могут открываться и закрываться. Можно создавать *формы* как объекты класса *Form*. Однако чаще всего создается специальный класс *FormX* – наследник класса *Form.*

Так в Windows-приложении, создаваемом по умолчанию, создается класс *Form1*. Если новые формы создается в приложении с использованием пункта меню *Add Windows Form,* то создаются классы *Form2, Form3* и т.д. Каждая новая форма в проекте – это объект собственного класса.

Форма может быть **модальной** или **немодальной** в зависимости от того, каково ее окно. Если открытое окно **не** позволяет временно переключиться на работу с другим открытым окном, окно называется модальным. Метод *ShowDialog* открывает форму как модальную. Основное назначение модальных окон – организация диалога с пользователем. Пока диалог не завершен, покинуть форму не удастся.

Немодальные окна допускают одновременную работу в нескольких открытых окнах. Метод *Show* открывает форму как немодальную.

Обмен информацией между формами

Часто многие формы должны работать с одними и теми же объектами, производя над ними различные операции. Обычно объект создается в одной из *форм* (чаще всего в главной).

Пр создании следующей *формы* глобальный объект передается конструктору новой *формы* в качестве аргумента. Одно из полей новой *формы* должно представлять собой ссылку на объект соответствующего класса. В конструкторе остается только связать ссылку с переданным конструктору аргументом.

Если глобальный объект создается в главной *форме*, то в качестве аргумента можно передавать не сам объект, а содержащий его контейнер – *главную форму*.

В нашей задаче мы хотим использовать текст, содержащийся в компоненте *textBoxInput*, расположенном на первой форме, во второй форме. Для этого нужно эти две формы связать друг с другом.

В модуле *Form1.cs* объявим переменную `Form2 FormDictionaries;`

```
namespace WinDictionaries
{
    public partial class Form1 : Form
    {
       Form2 FormDictionaries;

        public Form1()
        {
            InitializeComponent();
        }
```

В обработчике события, в котором вызывается вторая форма, (например, нажатие на кнопку) напишем следующий код:

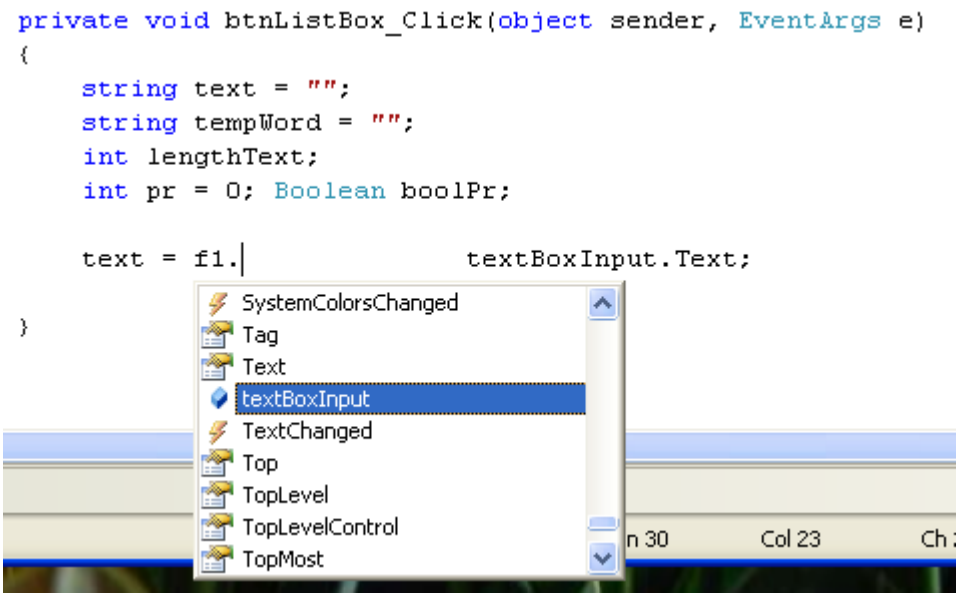
```
      private void btnDictionry_Click(object sender, EventArgs e)
        {
            FormDictionaries = new Form2(this);

            FormDictionaries.Show();
        }
```

В модуле *Form2.cs* вносим изменения в конструктор по умолчанию:

```
namespace WinDictionaries
{
    public partial class Form2 : Form
    {
       Form1 f1;    //Создаем переменную, для хранения ссылки на первую форму

       public Form2(Form1 refForm1) //Меняем конструктор - указываем параметр
                                    //- имя для ссылки на первую форму
        {
          InitializeComponent();
          f1 = refForm1;      //переменной f1 присваиваем значение ссылки
        }
      . . .
```

Теперь, когда вторая форма связана с первой формой – контейнером, содержащим нужные нам в проекте объекты, эти объекты нужно сделать доступными из второй формы. Для этого следует сменить значение свойства *Modifiers* (в разделе *Design* в окне *properties*) с *private* на *public* для этих объектов. Например, так как по условию задачи текст для словарей, создаваемых на второй форме, должен быть взят из компоненты *textBoxInput*, находящейся на первой форме, у него следует изменить значение свойства *Modifiers* (в разделе *Design* в окне *properties*) с *private* на *public*.

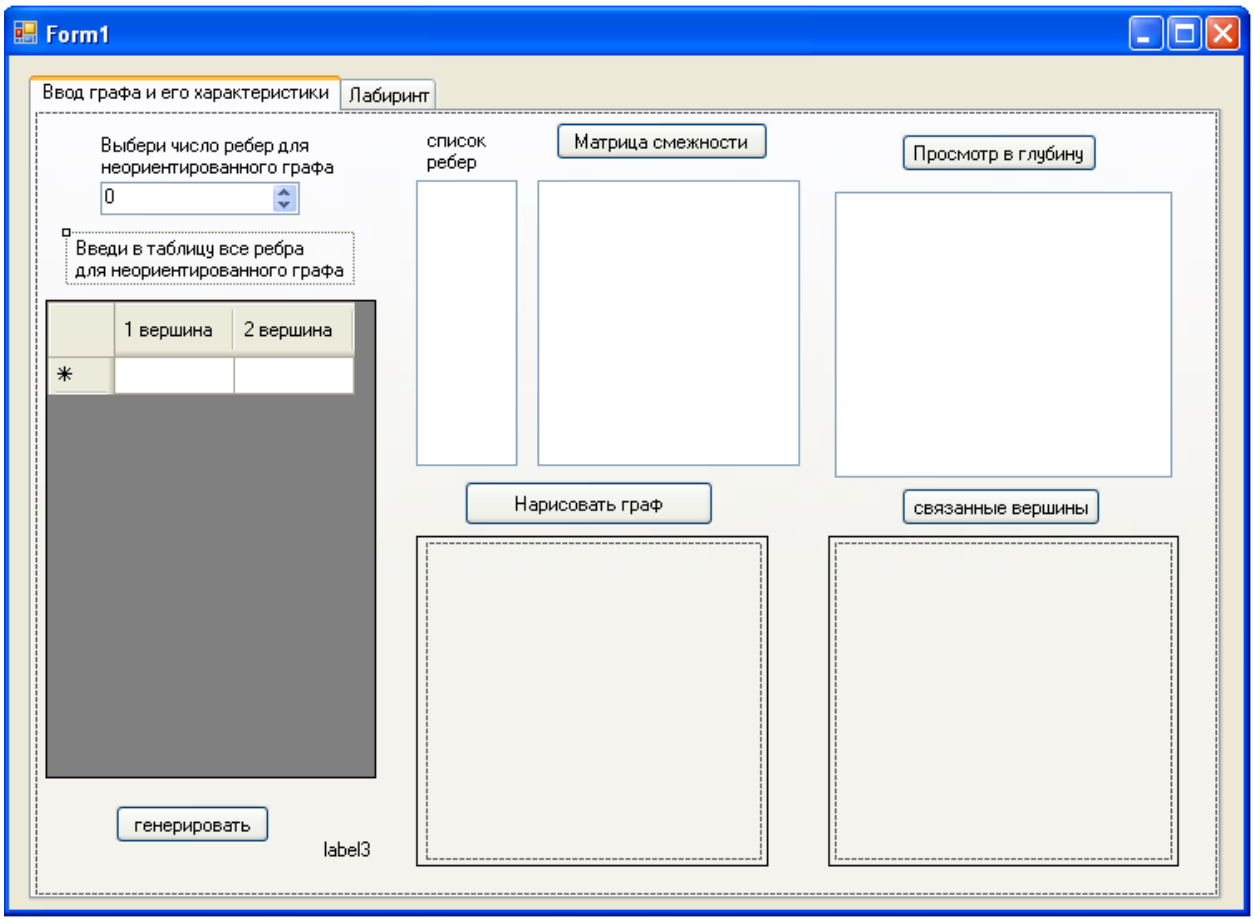
Тогда этот объект становится доступным в обработчиках событий на форме *Form2*.


private void btnListBox_Click(object sender, EventArgs e)
{
string text = "";
string tempWord = "";
int lengthText;
int pr = 0; Boolean boolPr;
text = f1.
textBoxInput.Text;
}
SystemColorsChanged
Tag
Text
textBoxInput
TextChanged
Top
TopLevel
TopLevelControl
TopMost
n 30
Col 23

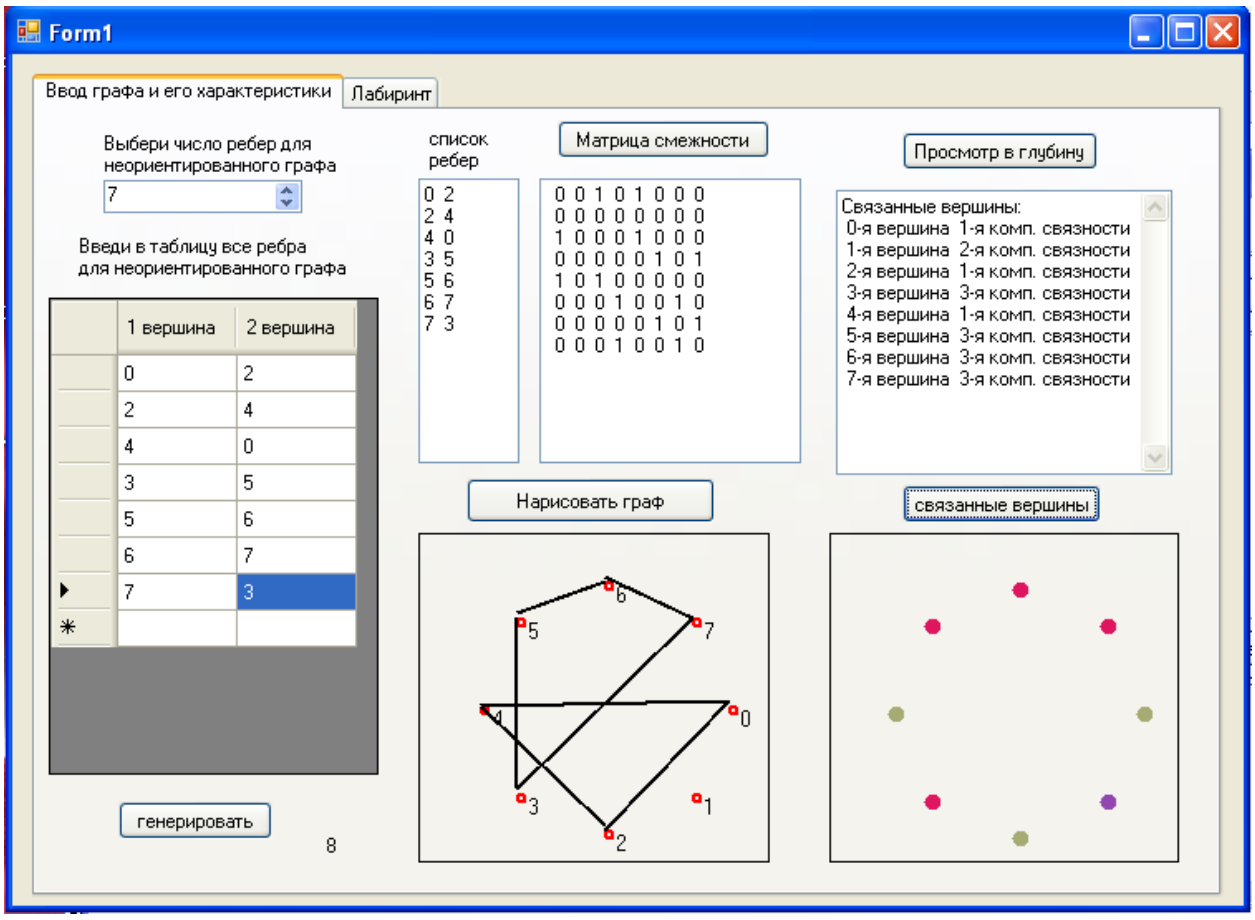
9. Алгоритмы на графах

1. Рекомендуем начать работу с создания формы, приведенной ниже на первом рисунке. Форма содержит следующие компоненты:

- **TabControl** – для создания двух страниц. На одной вводится граф, и определяются его характеристики, на другой строится лабиринт.
- **DataGridView** – для создания таблицы, в которую будут вводиться и из которой будут читаться ребра графа.
- **NumericUpDown** – для выбора числа ребер графа.
- Несколько компонент **TextBox** для вывода результатов работы программы.
- Две компоненты **PictureBox**, помещенных на компоненты **Panel**, для вывода графических изображений.
- Необходимые метки и кнопки, для которых будут написаны соответствующие обработчики.



Ниже приведен образец формы в работающем состоянии.



2. Удобно при описании всех переменных и массивов, необходимых в задаче, для ребер графа использовать **структуру**

```
struct myRibs
{
    public int beginRib; //Первая вершина ребра
    public int endRib;   //Вторая вершина ребра
}
```

Сначала достаточно описать массив ребер arrRibs, как массив структур, целочисленный двумерный массив для представления матрицы смежности (mSm), переменную для числа вершин (nNodes) в графе и переменную для числа ребер (countRibs) в графе.

```
myRibs[] arrRibs = new myRibs[50];
```

. . .

3. Написание нужных процедур следует начать с процедур ввода ребер графа. По условию задачи ребра можно либо вручную набирать в ячейках таблицы, либо вводить из заранее подготовленных файлов. В данных указаниях рекомендуется написать процедуру автоматического генерирования ребер, которая будет работать при нажатии на кнопку «Генерировать» после выбора из элемента NumericUpDown нужного числа ребер. Для генерирования ребер используется класс Random.

Ниже приведены заготовки для процедур генерирования ребер и определения характеристик графа.

```
//генерируем неориентированный граф
private void btnGen_Click(object sender, EventArgs e)
{
    decimal countRibs =  читаем число ребер их ComboBox
    dataGridView1.RowCount =  Задаем число строк в DataGridView
    Random num = new Random();
    int number1;
    for (int i = 0; i < countRibs; i = i + 1)
    {
       Генерируем пару случайных чисел для вершин ребра, исключая петли
       и записывем их в нулевой и первый столбцы таблицы соответственно
    }
}

//определяем характеристики неориентированного графа (число вершин и
//матрицу смежности)
//нажатие на кнопку «Матрица смежности»
private void btnAdd_Click(object sender, EventArgs e)
    {
        string strVal = "";

        for (int i = 0; i < countRibs; i++)
        {
            Переписываем значения из таблицы в массив ребер arrRibs
        }
        выводим массив ребер в TextBox1
        считаем число вершин  nNodes в графе по списку ребер, находя
        максимальный номер вершины
        Строим и выводим матрицу смежности, учитывая, что граф
        неоринтированный
     }
```

4. Теперь, когда характеристики графа определены, можно приступить к написанию процедур для рисования графа. Рекомендуется вершины графа располагать равномерно по окружности, а угловые расстояния между ними определять деля полный угол на число вершин.

Для написания процедуры рисования графа потребуется сделать дополнительные описания переменных

```
        float radius;
        float widthX;
        float heightY;
        float x;    float y;
        int nom = 0;
        Point pNode;
        Image image1;    //Экземпляр класса Image для рисунка графа

//определение области для рисунка графа
image1 = new Bitmap(pictureBox1.Width, pictureBox1.Height);
```

Удобно для определения места расположения очередной вершины графа написать отдельную функцию.

```
//Определение позиции k-ой вершины для размещения по кругу
public float getNodePos(int k)
{
    double ugol = (2 * Math.PI) / nNodes * k;
    x = (float)(radius * Math.Cos(ugol) + widthX);
    y = (float)(radius * Math.Sin(ugol) + heightY);
    return x;
}
```

Для рисования графа также лучше написать специальную процедуру, которую затем вызывать при нажатии кнопки «Нарисовать граф»

```
//Процедура рисования графа
  private void Draw()
  {
      image1 = new Bitmap(pictureBox1.Width, pictureBox1.Height);
      Pen pen = new Pen(Color.Red, 2);
      Graphics g = Graphics.FromImage(image1);
      widthX = pictureBox1.Width / 2;
      heightY = pictureBox1.Height / 2;
      radius = Math.Min(widthX, heightY) - 20;

      for (int i = 0; i < nNodes; i++) //рисуем кружки-вершины
      {
          pNode = new Point();
          getNodePos(i);
          pNode.X = Convert.ToInt32(x);
          pNode.Y = Convert.ToInt32(y);
          g.DrawEllipse(pen, pNode.X + 5, pNode.Y + 5, 5, 5);
          g.DrawString(i.ToString(), new Font("new Times Roman", 10),
                Brushes.Black, pNode.X + 10, pNode.Y + 5);
      }

     //рисуем ребра, просматривая матрицу смежности и учитывая, что в
     //матрице смежности каждое ребро повторено дважды, т.к. граф
     //неориентированный.
                 .  .  .

      pictureBox1.Image = image1;
  }


  private void btnDraw_Click(object sender, EventArgs e)
  {
      Draw();
  }
```

5. Следующим этапом работы является написание алгоритмов для определения смежных вершин в графе, компонент смежности и их вывода в графическое окно с раскраской элементов каждой компоненты своим цветом.

Для выделения компонент связности может быть применен алгоритм **просмотра графа вглубь**. В этом случае алгоритм формирует элементы массива **C,** определяющие компоненты связности для вершин графа.

`C[i] = q`, если `i`-я вершина принадлежит `q`-й компоненте связности. Первоначально всем элементам массива `C` присваиваются нули, что означает, что ни одна из вершин графа не приписана ни к одной компоненте. Затем рекурсивный алгоритм просмотра графа вглубь находит очередную вершину, для которой `C[i] = 0`, и приписывает этой вершине номер новой компоненты, после чего производит просмотр графа, начиная с этой вершины, присваивая всем связанным с данной вершиной один и тот же номер.

Ниже приведен вариант написания такого алгоритма.

```
private void findCompLink(int k, int q)    //обход в глубину
{
    for (int i = 0; i < nNodes; i++)
    {
        if (Msm[k, i] == 1 || Msm[i, k] == 1)
        {
            if (compLink[i] == -1)
            {
                compLink[i] = q;
                findCompLink(i, q);
            }
        }
    }
}
```

При записи алгоритма использовались имена:

`Msm` – матрица смежности графа,

`k` – номер вершины , с которой начинается поиск,

`compLink` – массив компонент связности,

`q` - номер компоненты связности,

`nNodes` – число вершин в графе.

6. Отображение компонент связности производится в свойстве Image PictureBox2 (по аналогии с тем, как это делалось при рисовании графа).

Ниже приведены фрагменты соответствующего кода с комментариями.

```
//Отображение компонент связности
private void graphCompLinked()
{
    Graphics g = Graphics.FromImage(image2);
    Pen penlink = new Pen(Color.Black, 2);
```

```
        widthX = pictureBox2.Width / 2;
        heightY = pictureBox2.Height / 2;
        radius = Math.Min(widthX, heightY) - 20;
        int i;   int q = 0;
        compLink = new int[nNodes];

       //Начальное  присваивание значений элементам массива compLink
             .  .  .
        //Алгоритм просмотра графа вглубь
             .  .  .
        //далее идет отображение компонент связности
        //Нужно не забыть описать соответствующий Image2 и выделить для
        //него область в PictureBox2.
             .  .  .
        //описать и сформировать массив цветов для компонент связности.
        //для удобства следует самостоятельно написать функцию
        //randomColor(), случайно формирующую цвет из R, G, B
       Color[] cLink = new Color[kCompLink];
       for (int k = 0; k < kCompLink; k++)
        {cLink[k] = randomColor();}

       Color[] clVersh = new Color[nNodes];
        //clVersh - массив цветов для окраски всех вершин. Цвет вершины
        //определяется тем, к какой компоненте связности она относится.
        //Компонента связности для к-й вершины находится в к-ом элементе
        // массива compKink
       for (int k = 0; k < nNodes; k++)
         {clVersh[k] = cLink[compLink[k]];}

        // вывод связанных вершин в текстовое окно
             .  .  .

        //раскраска вершин по компонентам связности
        for (int j = 0; j < nNodes; j++)
        {
            Point pVersh = new Point();
            getNodePos(j);
            pVersh.X = Convert.ToInt32(x);
            pVersh.Y = Convert.ToInt32(y);
            penlink = new Pen(clVersh[j], 2);
            SolidBrush myBrush = new SolidBrush(clVersh[j]);
            g.FillEllipse(myBrush, pVersh.X + 5, pVersh.Y + 5, 10, 10);
            //g.DrawEllipse(penlink, pVersh.X + 5, pVersh.Y + 5, 5, 5);
        }
        pictureBox2.Image = image2;
    }
```

7. Следующим этапом работы является написание алгоритма поиска пути в лабиринте с использованием просмотра графа вширь.

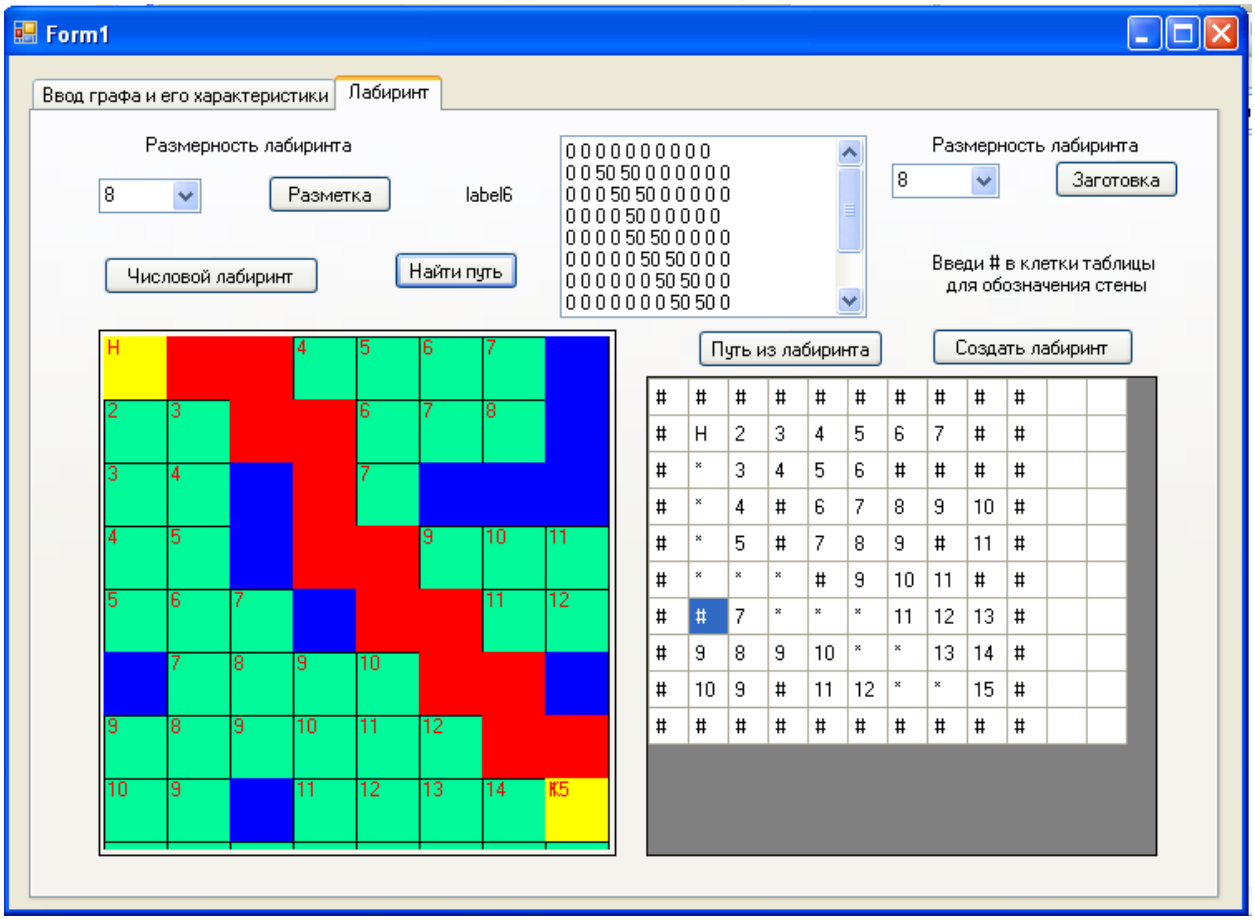
Лабиринт задан в виде квадратной таблицы. Клетка таблицы, имеющая нулевое значение, соответствует проходу в лабиринте, клетка с положительным значением – стена. Левая верхняя клетка является началом пути по лабиринту, а правая нижняя клетка – конец пути. Используя метод просмотра графа в ширину, найти кратчайший путь в лабиринте или определить, что пути не существует.

Входные величины: таблица, представляющая лабиринт, заполняется на форме вручную.

Результаты выводятся на форму в графическое окно в виде сетки, отображающей квадратную таблицу. Ячейки-стены закрашены в черный цвет. Ячейки, по которым проходит кратчайший путь, последовательно пронумерованы. Номера выводятся в ячейках ярким цветом. Остальные ячейки не закрашены.

Естественно, сначала нужно изучить алгоритм просмотра графа вширь.

На ниже приведенной форме приведен образец работы алгоритма. Используется два способа задания лабиринта и отображения пути. Первый – лабиринт представлен с помощью таблицы `DataGridView`, значения ячеек которой могут задаваться вручную (стена – символ #). Найденный в результате просмотра путь из начальной вершины в конечную вершину отображается символом * в соответствующих ячейках таблицы.



Второй способ – представление лабиринта на графическом поле элемента PictureBox с выделением стен и клеток пути выбранным заранее цветом.

В первом случае в таблице DataGridView краевые элементы таблицы отображены как стенки, во втором случае при отображении они отсуствуют.

На форме присутствует элемент TextBox, в который можно выводить значения меток вершин при просмотре графа. На приведенном рисунке в TextBox отображен с помощью некоторого символа (числа 50) найденный путь для графа, представленного в графическом окне.

В приведенном ниже фрагменте кода при написании алгоритма просмотра графа вширь для лабиринта, использующего два описанных представления лабиринта, описаны следующие глобальные переменные:

```
int razm;         //размер лабиринта для таблицы DataGridView
int razmLab;      //размер лабиринта для графического окна PictureBox
//массив для числового представления лабиринта, заданного в таблице
int[,] L = new int[20, 20];
//массив для числового представления лабиринта, заданного в графическом
//окне
int[,] Labir = new int[20, 20];
//графическое окно для лабиринта, если оно используется в нескольких
//процедурах
Graphics gLabir;
```

Алгоритм поиска пути в лабиринте для представленного образца формы в первом случае возможно потребует написания следующих процедур:

- Создать в DataGridView незаполненный стенами лабиринт, в котором выделены только границы и начальная и конечная клетки;
- После ручного построения лабиринта получить его числовое представление в массиве **L** и провести первичную разметку вершин лабиринта (ячеек таблицы). Отобразить эту разметку в ячейках таблицы.
- Реализовать любой алгоритм поиска кратчайшего пути в размеченном лабиринте.

Для второго случая, когда лабиринт представлен в графическом окне, для выполнения задания потребуется добавить работу с графикой в PictureBox.

- Нарисовать лабиринт в виде незаполненных клеток, добавить массив прямоугольников (например, Rectangle[,] arrrect), который позволит работать на графическом поле с отдельной клеткой лабиринта.

- Написать процедуру для отображения стенки в клетке на графическом поле при нажатии мышкой на эту клекту.
- После ручного построения лабиринта получить его числовое представление в массиве **`Labir`** и провести первичную разметку вершин лабиринта (клеток). Отобразить эту разметку в клетках лабиринта.
- Пользуясь первичной разметкой, реализовать любой алгоритм поиска кратчайшего пути в размеченном лабиринте. Оформить графический вывод в `PictureBox`.

10. Алгоритмы линейной алгебры

Цель работы – реализация алгоритмов решения некоторых задач линейной алгебры на основе представления векторов и матриц как объектов.

ЗАДАНИЕ 1. В отдельном модуле создать класс для работы с векторами. Форма проекта должна содержать компоненты, позволяющие организовать просмотр и вычисления двух разных векторов одновременно.

При создании класса следует учесть перечисленные ниже требования.

Минимальный набор полей: количество элементов, значения элементов.

Минимальный набор методов: конструктор, деструктор, копирование вектора, получение размеров вектора, вычисление модуля вектора, умножение вектора на число.

Минимальный набор свойств: ввод и вывод значения элемента вектора.

ЗАДАНИЕ 2. В отдельном модуле создать класс для работы с квадратными матрицами. Форма проекта должна содержать компоненты, позволяющие организовать просмотр и вычисления двух разных матриц одновременно.

При создании класса следует учесть перечисленные ниже требования.

Минимальный набор полей: размер матрицы, значения элементов.

Минимальный набор методов: конструктор, деструктор, копирование матрицы, получение размеров матрицы, транспонирование матрицы, вычисление определителя.

Минимальный набор свойств: ввод и вывод значения элемента матрицы.

1. Объявление класса

Класс определяет характеристики и поведение объектов данного класса. Его можно считать заготовкой или шаблоном, на основе которого создаются **объекты** данного класса. В C# характеристики объектов хранятся в переменных, называемых **полями** данного класса, а моделирование поведения выполняется **методами**, которые представляют собой набор операторов, выполняющих некоторую задачу. Поля и методы называют **компонентами** класса.

Объявление класса осуществляется при помощи ключевого слова **`class`** и имеет следующий синтаксис:

```
[модификатор доступа] class имя-класса {тело класса}
```

Классы обычно объявляются с модификатором доступа `public`, который означает, что класс доступен везде и без ограничений.

2. Создание объекта

Создание объекта также называют **созданием экземпляра** класса, а созданный объект, соответственно, **экземпляром** класса.

Для создания объекта применяется оператор `new` и специальный метод, называемый **конструктором**. Наш класс `MyVector` позволяет создавать экземпляры следующим образом:

```
MyVector V1;
V1 = new MyVector();
```

В первом предложении создается **ссылка** на объект типа `MyVector`.

Во втором предложении сначала работает оператор `new`, который выделяет память для хранения объекта, а затем конструктор с именем `MyVector()` создает непосредственно сам объект. Обратим внимание, что имя конструктора всегда совпадает с именем класса.

В приведенном примере используется конструктор по умолчанию, который доступен всегда. Этот конструктор не принимает никаких параметров. Он только создает объект и не выполняет никаких других действий. Если возникает необходимость в каких-то дополнительных действиях при создании объекта, можно создавать собственный конструктор для класса. Чаще всего дополнительные действия – инициализация полей вновь созданного объекта. Значения этих полей передаются конструктору в виде параметров.

Заметим также, что конструктор не может возвращать никаких значений.

Созданный объект является **ссылочным** типом. В C# ссылочные типы, такие как объекты и строки, хранятся в управляемой области памяти, называемой «**кучей**». В обычной памяти (стеке) хранятся только ссылки на объекты. В C# имеется специальный объект, который называется «сборщик мусора». Его задача – удалять из памяти данные, которые больше не используются в процессе работы. Такие данные автоматически ставятся сборщиком мусора в очередь на удаление. В C# обычно можно не заботиться об очистке памяти и предоставить эту работу сборщику мусора. В тех случаях, когда перед удалением объекта нужно выполнить некоторые действия, используются специальные объекты, называемые **деструкторами**.

3. Указатель `this`

Во всех методах класса доступна неявная переменная **`this`**, представляющая собой указатель на тот экземпляр класса, который был использован при вызове этого метода. Переменная **`this`** передается компилятором методу в качестве скрытого параметра. Статические методы не имеют указателя **`this`**, так как их нельзя связать ни с каким объектом, а только с самим классом. Указатель **`this`** может использоваться для доступа к членам класса в конструкторах или нестатических методах.

Рассмотрим пример объявления класса «Вектор».

```
public class MyVector
{
    //Класс имеет два поля - число компонент и массив значений
    private int n = 0;
    private int[] elements;
      ...

}
```

В методах, где используется поле `n`, оно может быть записано в виде уточняющего имени: `this.n`.

Указатель **`this`** используется в методах для объявления **индексаторов**. Ниже приведен пример определения **свойств** (properties) для нашего класса.

```
      //Определяем свойство - длину вектора

      public int lengthVector
      {
          get
          {
              return n;
          }
          set
          {
              this.n = value;
              elements = new int[n];
          }
       }



       //определяем индексированное свойство для поля – массива компонент

       public int this[int index]
       {
           get
           {
               return elements[index];
           }
           set
           {
               elements[index] = value;
           }
       }
            ...
     }
```

4. Методы

Методы представляют собой процедуры и функции, принадлежащие классу. Можно сказать, что методы определяют поведение класса. В классе обязательно должен присутствовать метод, называемый **конструктор,** который отвечает за создание объекта, выделение для него необходимой памяти и необходимую инициализацию полей**.** Наличие деструктора не обязательно, т.к. происходит автоматическая сборка мусора и после того, как

метод будет выполнен, все связанные с ним ресурсы будут освобождены автоматически.

При проектировании класса можно создать произвольное число методов, нужных для решения конкретных задач. Это относится также и к конструкторам.

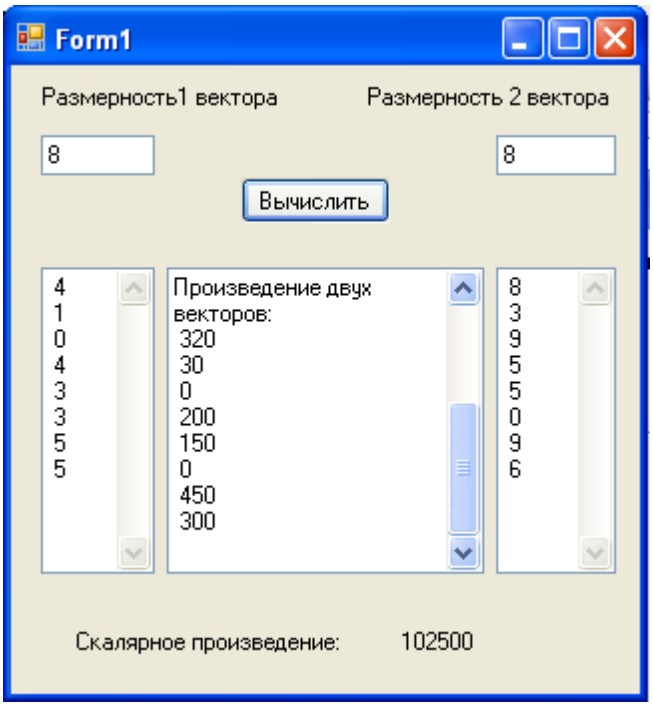
Методы могут быть описаны как невиртуальные, виртуальные или статические, для чего к ним добавляются соответствующие директивы (static, virtual, override, overload).

5. Пример создания класса

Первым заданием лабораторной работы является создание класса для работы с векторами. Ниже приведен фрагмент программы для создания класса и возможный вариант формы для выполнения заданий лабораторной работы.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace WinAppVector
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        public class MyVector
        {
            //Класс имеет два поля - число компонент и массив значений
            private int n = 0;
            private int[] elements;

            //Определяем свойство - длину вектора
            public int lengthVector
            {
                get
                {
                    return n;
                }
                set
                {
                    this.n = value;
                    elements = new int[n];
                }
            }

            //Определяем свойство для обращения к массиву значений
            public int this[int index]
            {
                get
                {
```

```
            return elements[index];
        }
        set
        {
            elements[index] = value;
        }
    }

    //Метод умножения элементов вектора на коэффициент
    public int[] product(int factor)
    {
        for (int i = 0; i < lengthVector; i++)
        {
            elements[i] = this[i] * factor;
        }
        return elements;
    }
    ...
```




```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
```

```
namespace WinAppVector
{
   public partial class Form1 : Form
   {
      public Form1()
      {
         InitializeComponent();
      }
      public class MyVector
      {
         //Класс имеет два поля - число компонент и массив значений

         private int n = 0;
         private int[] elements;

         //Определяем свойство - длину вектора
         public int lengthVector
         {
            get
            {
               return n;
            }
            set
            {
               this.n = value;
               elements = new int[n];
            }
         }
         //Определяем свойство для обращения к массиву значений
         public int this[int index]
         {
            get
            {
               return elements[index];
            }
            set
            {
               elements[index] = value;
            }
         }

         public int[] product(int factor)
         {
            for (int i = 0; i < lengthVector; i++)
            {
```

```
            elements[i] = this[i] * factor;
        }
        return elements;
    }

    public int[] product(MyVector secondVector)
    {
        for (int i = 0; i < lengthVector; i++)
        {
            elements[i] = this[i] * secondVector[i];
        }
        return elements;
    }


    public int scalarProduct(MyVector secondVector)
    {
        int res = 0;
        for (int i = 0; i < lengthVector; i++)
        {
            res = res + this[i] * secondVector[i];
        }
        return res;
    }
}



private void button1_Click(object sender, EventArgs e)
{
    Random num;
    num = new Random();

    MyVector V1 = new MyVector();

    V1.lengthVector = Convert.ToInt32(textBox3.Text);
    for (int i = 0; i < V1.lengthVector; i++)
    {
        V1[i] = num.Next(0, 10);
    }
    string sOut1 = "";
    for (int i = 0; i < V1.lengthVector; i++)
    {
        sOut1 = sOut1 + " " + V1[i].ToString() + "\r\n";
    }
```

```
            textBox1.Text = sOut1;

            MyVector V2 = new MyVector();
            V2.lengthVector = Convert.ToInt32(textBox4.Text);
            for (int i = 0; i < V2.lengthVector; i++)
            {
               V2[i] = num.Next(0, 10);
            }
            string sOut2 = "";
            for (int i = 0; i < V2.lengthVector; i++)
            {
               sOut2 = sOut2 + " " + V2[i].ToString()+ "\r\n";
            }
            textBox2.Text = sOut2;

            V2.product(10);
            string sOut3 = "";
            for (int i = 0; i < V2.lengthVector; i++)
            {
               sOut3 = sOut3 + " " + V2[i].ToString() + "\r\n";
            }
            textBox5.Text = sOut3;

            V1.product(V2);
            sOut3 = sOut3 + "Произведение двух векторов:" + "\r\n";
            for (int i = 0; i < V1.lengthVector; i++)
            {
               sOut3 = sOut3 + " " + V1[i].ToString() + "\r\n";
            }
            textBox5.Text = sOut3;

            //Скалярное произведение двух векторов
            int result = 0;
            result = V1.scalarProduct(V2);
            label3.Text = result.ToString();

        }
    }
}
```