# Лабораторная работа №16

**ХАРЬКОВСКИЙ НАЦИОНАЛЬНЫЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И МЕХАТРОНИКИ**

**Кафедра информационных технологий и мехатроники**

# МЕТОДИЧЕСКИЕ УКАЗАНИЯ

**по проведению лабораторных работ по дисциплине «Алгоритмизация и программирование»**
**для студентов специальности 6.050101 “Компьютерные науки”**

Разработчик - доцент кафедры информационных технологий и мехатроники
кандидат технических наук, старший научный сотрудник
Тимонин Владимир Алексеевич

Харків 2015

# Лабораторная работа №16
# Исследование возможностей интегрированной среды разработки Visual C# для создания приложений по обработке бинарных файлов.

**Цель работы –** исследовать возможности интегрированной среды разработки Visual Studio 2010 и получить практические навыки по созданию приложений по обработке бинарных файлов.

## 1. Теоретические сведения
## Считывание и запись двоичных данных

В C# предусмотрены классы, которые позволяют считывать содержимое файлов и записывать в них информацию. Так как на уровне операционной системы все файлы обрабатываются на побайтовой основе, то в C# определены методы, предназначенные для считывания байтов из файла и записи байтов в файл. Файловые операции, ориентированные на символы, используются в случае текстовых файлов.

Чтобы создать байтовый поток с привязкой к файлу, используется класс FileStream, являющийся производным от Stream и обладающий функциональными возможностями базового класса. Так как потоковые классы, включая FileStream, определены в пространстве имен System.IO, то при их использовании в начало программы необходимо включить оператор:

**using System.IO;**.

До сих пор мы считывали и записывали байты или символы, но эти операции ввода-вывода можно выполнять и с другими типами данных. Например, вы могли бы создать файл, содержащий int-, double- или short-значения. Для считывания и записи двоичных значений встроенных C#-типов используйте классы BinaryReader и BinaryWriter. Важно понимать, что эти данные считываются и записываются с использованием внутреннего двоичного формата, а не в текстовой форме, понятной человеку.

### 1.1. Запись двоичных данных в файл

Класс BinaryWriter представляет собой оболочку для байтового потока, которая управляет записью двоичных данных. Его наиболее употребительный конструктор имеет следующий вид:

**BinaryWriter(Stream outputStream);**

где параметр **outputStream** означает поток, в который будут записываться данные. Чтобы записать выходные данные в файл, можно использовать для этого параметра объект, созданный классом FileStream. Если поток **outputStream** имеет **null**-значение, генерируется исключение типа ArgumentNullException, а если поток **outputStream** не открыт для записи, — исключение типа ArgumentException.

В классе BinaryWriter определены методы, способные записывать значения всех встроенных C#-типов (некоторые из них перечислены в табл. 1). В классе BinaryWriter также определены стандартные методы Close() и Flush().

Таблица 1. Методы класса BinaryWriter

| Метод | Описание |
|---|---|
| **void write (sbyte val)** | Записывает byte-значение (со знаком) |
| **void write (byte val)** | Записывает byte-значение (без знака) |
| **void write (byte [ ] buf)** | Записывает массив byte-значений |
| **void write (short val)** | Записывает целочисленное значение типа short |
| **void write (ushort val)** | Записывает целочисленное ushort -значение |
| **void write ( int val)** | Записывает целочисленное значение типа int |
| **void write (uint val)** | Записывает целочисленное uint-значение |
| **void write (long val)** | Записывает целочисленное значение типа long |

| | |
|---|---|
| **void write (uiong val)** | Записывает целочисленное ulong-значение |
| **void write (float val)** | Записывает float-значение |
| **void write (double val)** | Записывает double-значение |
| **void write (char val)** | Записывает символ |
| **void write (char [ ] buf)** | Записывает массив символов |
| **void write (string val)** | Записывает string-значение с использованием его внутреннего представления, которое включает спецификатор длины |

**Пример 1.** Нижеприведенный пример демонстрирует возможность записи в файл “test.dat” данных различных типов с использованием класса BinaryWriter.

```
BinaryWriter dataOut;
int i = 10;
double d = 1023.56;
bool b = true;
try
{
   dataOut = new BinaryWriter(new FileStream("test.dat", FileMode.Create));
}
catch (IOException exc)
{
   Console.Write(exc.Message);
   return;
}
try
{
   dataOut.Write(i);
   dataOut.Write(d);
   dataOut.Write(b);
   dataOut.Write(12.2 * 7.4);
 }
 catch (IOException exc)
 {
   Console.Write(exc.Message);
 }
 dataOut.Close();
```

### 1.2. Считывание двоичных данных из файла

Класс BinaryReader представляет собой оболочку для байтового потока, которая управляет чтением двоичных данных. Его наиболее употребительный конструктор имеет такой вид:

**BinaryReader(Stream inputStream);**

где параметр **inputStream** означает поток, из которого считываются данные. Чтобы выполнить чтение из файла, можно использовать для этого параметра объект, созданный классом FileStream. Если поток **inputStream** имеет **null**-значение, генерируется исключение типа ArgumentNullException, а если поток **inputStream** не открыт для чтения,— исключение типа ArgumentException.

В классе BinaryReader предусмотрены методы для считывания всех простых C#-типов (некоторые из них перечислены в табл. 2). При обнаружении конца потока все эти методы генерируют исключение типа EndOfStreamException, а при возникновении ошибки — исключение типа IOException.

Таблица 2. Методы класса BinaryReader

| Метод | Описание |
|---|---|
| **bool ReadBoolean()** | Считывает bool-значение |
| **byte ReadByte()** | Считывает byte-значение |
| **sbyte ReadSByte()** | Считывает sbyte-значение |
| **byte[] ReadBytes(int num)** | Считывает **num** байтов и возвращает их в виде массива |
| **char ReadChar()** | Считывает char-значение |
| **char[] ReadChars(int num)** | Считывает **num** символов и возвращает их в виде массива |
| **double ReadDouble()** | Считывает double-значение |
| **float ReadSingle ()** | Считывает float-значение |
| **short ReadIntl6()** | Считывает short-значение |
| **int ReadInt32()** | Считывает int-значение |
| **long ReadInt64()** | Считывает long-значение |
| **ushort ReadUIntl6()** | Считывает ushort-значение |
| **uint ReadUInt32()** | Считывает uint-значение |
| **ulong ReadUInt64()** | Считывает ulong-значение |
| **string ReadString()** | Считывает string-значение, представленное во внутреннем двоичном формате, который включает спецификатор длины. Этот метод следует использовать для считывания строки, которая была записана с помощью объекта класса BinaryWriter |

В классе BinaryReader также определены следующие модификации метода Read() (табл. 3). В случае неудачного исхода операции чтения эти методы генерируют исключение типаIOException.

Таблица 3. Модификации метода Read()

| Метод | Описание |
|---|---|
| **int Read()** | Возвращает целочисленное представление следующего доступного символа из вызывающего входного потока. При обнаружении конца файла возвращает значение -1 |
| **int Read (byte [ ] buf, int offset, int num)** | Делает попытку прочитать **num** байтов в массив **buf,** начиная с элемента **buf [offset]**, и возвращает количество успешно считанных байтов |
| **int Read (char [ ] buf, int offset, int num)** | Делает попытку прочитать **num** символов в массив **buf**, начиная с элемента **buf[offset],** и возвращает количество успешно считанных символов |

В классе BinaryReader также определен стандартный метод Close().

**Пример 2.** Нижеприведенный пример демонстрирует возможность считывания из файла "test.dat" данных различных типов с использованием класса BinaryReader.

```
BinaryReader dataIn;
int i;
double d, c;
bool b;
try
{
   dataIn = new BinaryReader(new FileStream("test.dat", FileMode.Open));
}
catch (FileNotFoundException exc)
```

```
{
   Console.Write(exc.Message);
   return;
}
try
{
   i = dataIn.ReadInt32();
   d = dataIn.ReadDouble();
   b = dataIn.ReadBoolean();
   c = dataIn.ReadDouble();
   Console.WriteLine(i.ToString() + "  " + d.ToString() + "  " +
                     b.ToString() + "  " + c.ToString());
}
catch (IOException exc)
{
   Console.Write(exc.Message);
}
dataIn.Close();
```

**Пример 3.** Нижеприведенное приложение «Наличие товара» реализует программу инвентаризации товара на складе. Для каждого вида товара приложение хранит соответствующее наименование, имеющееся в наличии количество и стоимость. Приложение предлагает пользователю ввести наименование товара, а затем выполняет поиск в базе данных. Если элемент найден, на экране отображается соответствующая информация. Один из вариантов результата решения задачи представлен на рис. 1.

```
BinaryWriter dataOut;
BinaryReader dataIn;
string item; // наименование элемента
int count;   // количество, имеющееся в наличии
double zena; // цена
try
{
   dataOut = new BinaryWriter(new FileStream("tovar.dat", FileMode.Create));
}
catch (IOException exc)
{
   Console.Write(exc.Message);
   return;
}
 // Записываем инвентаризационные данные в файл
try
{
   dataOut.Write("Компьютер");
   dataOut.Write(10);
   dataOut.Write(2784.95);
   dataOut.Write("Принтер");
   dataOut.Write(18);
   dataOut.Write(782.50);
   dataOut.Write("Сканер");
   dataOut.Write(5);
   dataOut.Write(418.15);
```

```
    dataOut.Write("Телевизор");
    dataOut.Write(8);
    dataOut.Write(3056.95);
}
catch (IOException exc)
{
     Console.Write(exc.Message);
}
dataOut.Close();
// Открываем файл инвентаризации для чтения информации
try
{
     dataIn = new BinaryReader(new FileStream("tovar.dat", FileMode.Open));
}
catch (FileNotFoundException exc)
{
    Console.Write(exc.Message);
    return;
}
// Поиск элемента, введенного пользователем
Console.Write(" Введите название товара ");
string what = Console.ReadLine();
Console.WriteLine();
try
{
    for (; ; )
    {
        // Считываем запись из базы данных
        item = dataIn.ReadString();
        count = dataIn.ReadInt32();
        zena = dataIn.ReadDouble();
        /* Если элемент в базе данных совпадает с элементом
           из запроса, то отображаем найденную информацию */
        if (item.CompareTo(what) == 0)
        {
            Console.WriteLine(" Товар <" + item + ">: " + count.ToString() +
                              " штук в наличии. Цена: " +  zena.ToString("C") +
                              " за каждую единицу.");
            Console.WriteLine(" Общая стоимость по наименованию <" +
                                item + "> " + (zena * count).ToString("C"));
            break;
        }
    }
}
catch (EndOfStreamException)
{
    Console.WriteLine(" Товар <" + what + "> отсутствует!");
}
catch (IOException exc)
{
    Console.Write(exc.Message);
}
```

```
dataIn.Close();
```

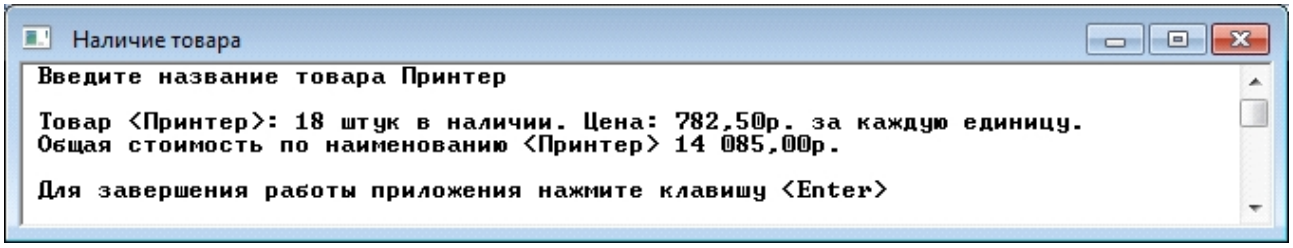


Рис. 1. Результат выполнения приложения«Наличие товара»

Необходимо обратить внимание на двоичный формат хранения информация о наличии товаров на складе. Следовательно, количество товаров, имеющихся в наличии, и их стоимость хранятся с использованием двоичного формата, а не в текстовом виде. Это позволяет выполнять вычисления над числовыми данными, не делая дополнительных преобразований.

Необходимо обратить ваше внимание на то, как обнаруживается здесь конец файла. Так как при достижении конца потока методы ввода двоичной информации генерируют исключение типа EndOfStreamException, то приложение просто считывает содержимое файла до тех пор, пока либо не найдет нужный элемент, либо не сгенерируется это исключение. Таким образом, для обнаружения конца файла в данном случае специального механизма не требуется.

### 1.3. Файлы с произвольным доступом

До сих пор мы использовали последовательные файлы, т.е. файлы, доступ к содержимому которых организован строго линейно, байт за байтом. Но в C# также возможен доступ к файлу, осуществляющийся случайным образом. В этом случае необходимо использовать методSeek(), определенный в классе FileStream. Этот метод позволяет установить индикатор позиции (или указатель позиции) в любое место файла.

Заголовочная информация о методе Seek() имеет следующий вид:

```
long Seek(long newPos, SeekOrigin origin);
```

где параметр **newPos** означает новую позицию, выраженную в байтах, файлового указателя относительно позиции, заданной параметром **origin**. Параметр **origin** может принимать одно из значений, определенных перечислением **SeekOrigin** (**Begin** - поиск от начала файла; **Current** – поиск от текущей позиции; **End** - поиск от конца файла).

После обращению к методу Seek() следующая операция чтения или записи данных будет выполняться на новой позиции в файле. Если при выполнении поиска возникнет какая-либо ошибка, генерируется исключение типа IOException. Если базовый поток не поддерживает функцию запроса нужной позиции, генерируется исключение типа NotSupportedException.

**Пример 4.** Нижеприведенное консольное приложение «Произвольный доступ» демонстрирует выполнение операций ввода-вывода с произвольным доступом. Приложение записывает в файл алфавит прописными буквами, а затем считывает значения через одно.

```
FileStream f;
string str = "";
char ch;
try
{
   f = new FileStream("alphavit.dat", FileMode.Create);
}
catch (IOException exc)
{
   Console.Write(exc.Message);
```

```
    return;
}
// Записываем в файл алфавит
for (int i = 0; i < 26; i++)
{
    try
    {
        f.WriteByte((byte)('A' + i));
    }
    catch (IOException exc)
    {
      Console.Write(exc.Message);
      return;
    }
}
try
{
    // Теперь считываем отдельные значения
    f.Seek(0, SeekOrigin.Begin); // Поиск первого байта
    ch = (char)f.ReadByte();
    Console.WriteLine(" Первое значение равно " + ch);
    f.Seek(1, SeekOrigin.Begin); // Поиск второго байта
    ch = (char)f.ReadByte();
    Console.WriteLine(" Второе значение равно " + ch);
    f.Seek(4, SeekOrigin.Begin); // Поиск пятого байта
    ch = (char)f.ReadByte();
    Console.WriteLine(" Пятое значение равно " + ch);
    // Теперь считываем значения через одно
    Console.Write(" Выборка значений через одно: ");
    for (int i = 0; i < 26; i += 2)
    {
        f.Seek(i, SeekOrigin.Begin); // Переход к i-му байту
        ch = (char)f.ReadByte();
        str += ch + "  ";
    }
    Console.WriteLine(" " + str);
}
catch (IOException exc)
{
    Console.Write(exc.Message);
}
f.Close();
```

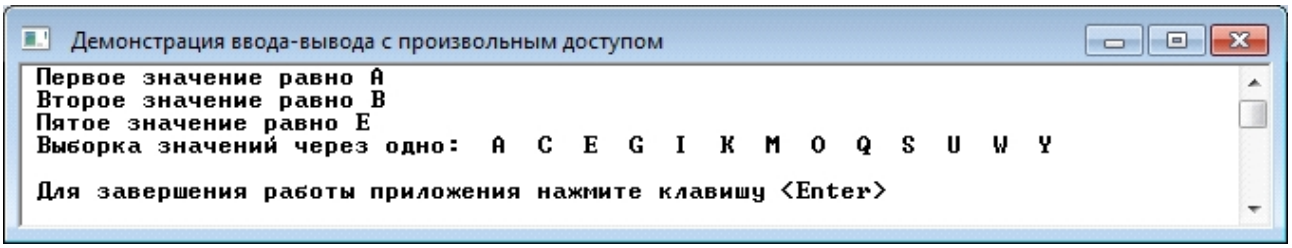


Рис. 2. Результат выполнения приложения«Произвольный доступ»

## 2. Рабочее задание

**Задание 1.** Руководствуясь теоретическим материалом раздела 1 изучить возможности языка C# по созданию приложений, обрабатывающие структуры данных, и выполнить практически все примеры, описанные в этом разделе.

**Задание 2.** Разработать приложение «Запись массива данных», с помощью которого можно записывать в файл «Result.dat» значения функции $R = \sum_{k=1}^{N} \frac{k}{k + \sqrt{1 + x^2}} e^{-x/k}$ для значений x, изменяющихся в диапазоне от $x_{нач}$ до $x_{кон}$ с шагом dx. Значения N, $x_{нач}$, $x_{кон}$ и dx вводятся пользователем с клавиатуры. В файл должны быть записаны данные в двоичном виде в последовательности **N, x, R** (см. рис.3).

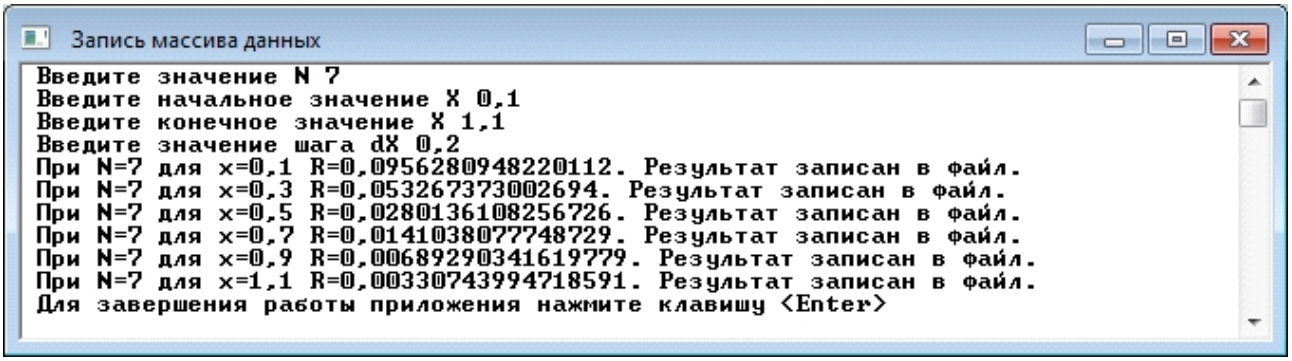


Рис. 3. Результат выполнения приложения «Запись массива данных»

**Задание 3.** Разработать приложение «Работа с массивами данных» модернизировав приложение «Запись массива данных» таким образом, чтобы при наличии данных в файле «Result.dat» можно было их отобразить на экране монитора, в противном случае вычислить значения функции **R** и записать их в файл.

## 3. Контрольные вопросы

1. Какой оператор необходимо использовать для работы с потоковыми классами?
2. Какие применяются методы для записи двоичных данных в файл?
3. Какие применяются методы для чтения двоичных данных из файла?
4. С помощью какого метода осуществляется произвольный доступ в файле?
5. Какие исключения могут генерироваться при работе с файлами?

## Литература


1. Голощапов А.Л. Microsoft Visual Studio 2010. – СПб.:БХВ-Петербург, 2011. – 544 с.: ил.
2. Культин Н.Б. Microsoft Visual C# в задачах и примерах. – СПб.: БХВ-Петербург, 2009. – 320 с.: ил.
3. Лабор В.В. Си Шарп: Создание приложений для Windows. – Мн.: Харвест, 2003. – 384 с.
4. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. Пер. с англ. - М.: «Русская Редакция», 2002.- 576 с.: ил.
5. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 2. Пер. с англ. - М.: «Русская Редакция», 2002.- 624 с.: ил.
6. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0. Пер. с англ. - М.: Издательский дом "Вильямс", 2011. — 1392 с.: ил.
7. Фаронов В.В. Программирование на языке C#. – СПб.: Питер, 2007. – 240 с.: ил.

8. Фленов М.Е. Библия C#. - СПб.: БХВ-Петербург, 2011. – 560с.: ил.