МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к лабораторной работе
«Работа с двоичными файлами в языке C#»
по дисциплине «Технологии программирования»

для студентов специальностей
122 – Компьютерные науки и информационные технологии,
124 – Системный анализ

Утверждено
редакционно-издательским
советом университета,
протокол № 2 от 23.06.16 г

Харьков
НТУ ”ХПИ”
2017

Методические указания к лабораторной работе «Работа с двоичными файлами в языке C#» по дисциплине «Технологии программирования» для студентов специальностей 122 – Компьютерные науки и информационные технологии, 124 – Системный анализ /Сост. Ю. Н. Кожин, О. Н. Малых, В. Ф. Прокопенков. – Харьков: НТУ “ХПИ”, 2017.– 40 с. – на рус.яз.

Составители: Ю. Н. Кожин,
О. Н. Малых,
В. Ф. Прокопенков,

Рецензент О.В. Горелый

Кафедра системного анализа и информационно-аналитических технологий

ВВЕДЕНИЕ

Язык C# и технология программирования .NET Framework пришли на смену языку C/C++ и обычному программированию для Windows. Возможности, предлагаемые платформой .NET, позволяют радикально облегчить жизнь программистов и разрабатывать программные приложения разного назначения.

Любая программа представляет собой закодированный алгоритм обработки данных, подчиненный цели решения определенной задачи. Во время исполнения программы вычислительной машиной её данные размещаются в оперативной памяти. Если программа завершает свою работу, то данные, размещаемые в оперативной памяти, будут потеряны. Данные могут быть сохранены во внешней памяти вычислительной машины. Сохранение данных необходимо выполнить, если они необходимы для пользователя программы или какой-либо программы.

Примером внешней памяти вычислительной машины является накопитель на жестком диске. Информация на диске организуется в виде файловой системы. Элементом в файловой системе является файл данных. Файл рассматривается как линейная последовательность байт с адресами, начиная со значения 0. Информация в файле, как и в памяти вычислительной машины, хранится в двоичном коде.

Существует два типа файлов – текстовые и двоичные, которые различаются по способу интерпретации хранимых в файле данных. Текстовые файлы удобны для пользователя программы, но двоичные файлы предпочтительны для программной обработки.

Предлагаемые методические указания помогут студентам познакомиться с возможностями языка C# для работы с двоичными файлами, необходимыми библиотечными классами платформы .NET Framework, и овладеть основами программирования двоичных потоков на языке C# как модели, используемой для работы с файлами.

Методические указания содержат необходимые теоретические сведения, а также рекомендации для выполнения лабораторных работ.

1. ПОТОКИ ДЛЯ РАБОТЫ С ДВОИЧНЫМИ ФАЙЛАМИ

В языке C# потоки данных (streams) являются объектами, из которых можно читать и в которые можно записывать данные. Они могут быть связаны с файлами, памятью и сетевыми коммуникациями. Одно из наиболее распространённых применений потоков – операции с файлами.

При разработке модели потоков учтены свойства операционных систем, встроенного программного обеспечения и устройств ввода\вывода, что избавляет программиста от необходимости учета низкоуровневых особенностей реализации. Потоки для работы с блоками двоичных данных основаны на базе абстрактного класса Stream (рис.1.1).

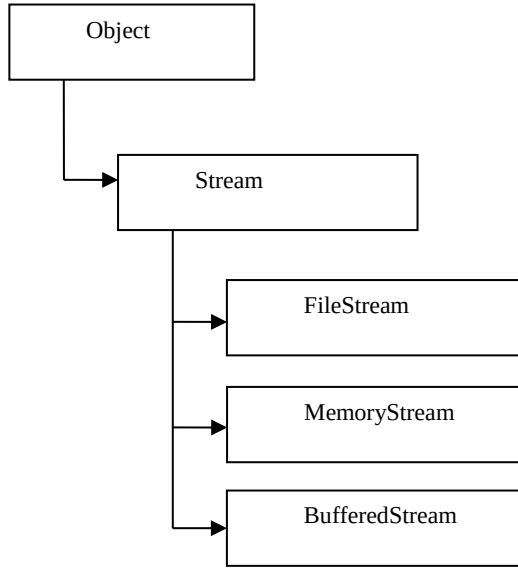

Рис.1.1. Иерархия классов для работы с блоками двоичных данных

Поток является абстракцией последовательности байтов файла для представления устройства ввода\вывода, канала связи и т.д. Потоки предусматривают три основные операции:

– чтение данных из потока как передачу данных из потока в массив байт;

– запись данных в поток как передачу данных из массива байт в поток;

– поиск данных в потоке как определение местонахождения (определение позиции размещения) последовательности двоичных данных в потоке.

В зависимости от назначения конкретного устройства связанный с ним поток может поддерживать или не поддерживать указанные операции.

1.1. Класс *Stream*

Класс Stream определяет необходимые свойства и методы, которые обеспечивают как синхронное, так и асинхронное (не требующее блокировки) взаимодействие со средой хранения данных (файлом, областью оперативной памяти, устройством ввода\вывода). Наиболее важные из них представлены в табл. 1.1.

Таблица 1.1 – Члены класса *Stream*

Член класса	Спецификация	Описание
CanRead	*public abstract bool CanRead { get; }*	Возвращает true, если поток поддерживает операции чтения
CanSeek	*public abstract bool CanRead { get; }*	Возвращает true, если поток поддерживает операции поиска
CanWrite	*public abstract bool CanWrite { get; }*	Возвращает true, если поток поддерживает операции записи
Length	*public abstract long Length { get; }*	Возвращает размер потока (блока двоичных данных) в байтах
Position	*public abstract long Position { get; set; }*	Устанавливает или возвращает позицию чтения\записи в потоке

Продолжение табл. 1.1

Close( )	*public virtual void Close( );*	Закрывает текущий поток и освобождает связанные с ним ресурсы
CopyTo( )	*public void CopyTo(Stream dest, int bufferSize);*	Читает все байты из текущего потока и записывает их в поток dest, используя буфер размером bufferSize>0 (по умолчанию 4096)
Flush( )	*public abstract void Flush( );*	Если используется буфер, записывает данные из буфера в связанный с потоком источник\ приемник данных и очищает буфер
Read( )	*public abstract int Read(byte[] buffer, int offset, int count);*	Считывает последовательность байт из потока, изменяя позицию чтения\записи. Если возвращаемое значение функции не ноль, то массив buffer в диапазоне индексов [offset ,offset + count] содержит считанную последовательность
ReadByte( )	*public virtual int ReadByte( );*	Считывает байт из потока, изменяя позицию чтения\записи. При успехе возвращает считанное значение как int, иначе -1 (признак конца потока)
Seek( )	*public abstract long Seek(long offset, SeekOrigin origin);*	Устанавливает позицию чтения\записи в потоке в положение offset (смещение в байтах) относительно позиции, которая определяется параметром origin (типа SeekOrigin): Begin – от начала потока, Current – от текущей позиции потока, End – от конца потока
SetLength( )	*public abstract void SetLength(long value);*	Устанавливает длину (размер двоичного блока данных) текущего потока
Write( )	*public abstract void Write(byte[]buffer, int offset, int count);*	Записывает последовательность байт из массива buffer в диапазоне индексов [offset , offset + count] в текущий поток, изменяя позицию чтения\записи потока
WriteByte( )	*public virtual void WriteByte(byte value);*	Записывает в текущий поток один байт из параметра value, изменяя позицию чтения\записи потока

1.2. Класс *FileStream*

Класс *FileStream* определяет абстрактные члены класса *Stream* и обеспечивает поток для чтения\записи двоичных файлов на диске. Режим использования файла определяется значениями типов перечислений *FileMode, FileAccess* и *FileShare* при открытии или создании файла методами классов *File, Filelnfo, FileStream*.

Пример программы работы с потоком *FileStream*:

```
using System;
using System.IO;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         // создаём объект файлового потока для работы с файлом
         FileStream myFStream = new FileStream("test.dat",
                                    FileMode.OpenOrCreate,
                                    FileAccess.ReadWrite);

         // записываем в файловый поток значения в диапазоне [0,255]
         for (int i = 0; i < 256; i++)
            myFStream.WriteByte((byte)i);

         // устанавливаем позицию чтения\записи потока на адрес 0
         myFStream.Position = 0;

         // считываем и выводим на консоль содержимое файлового потока
         for (int i = 0; i < myFStream.Length; i++)
         {
            Console.Write("{0:D3} ", myFStream.ReadByte( ));
```

```
            if ((i + 1) % 8 == 0)
            Console.WriteLine( );
        }
        myFStream.Close( );
    }
}
```

В данном примере открывается (или создается, если не существует) файл *test.dat* на чтение и запись, с которым в программе связывается поток – объект *myFStream.* В файл записывается последовательность байт, а затем читается с начала файла и выводится на консоль (рис.1.2). Если мы откроем созданный нами файл, то увидим результат на рис.1.3.

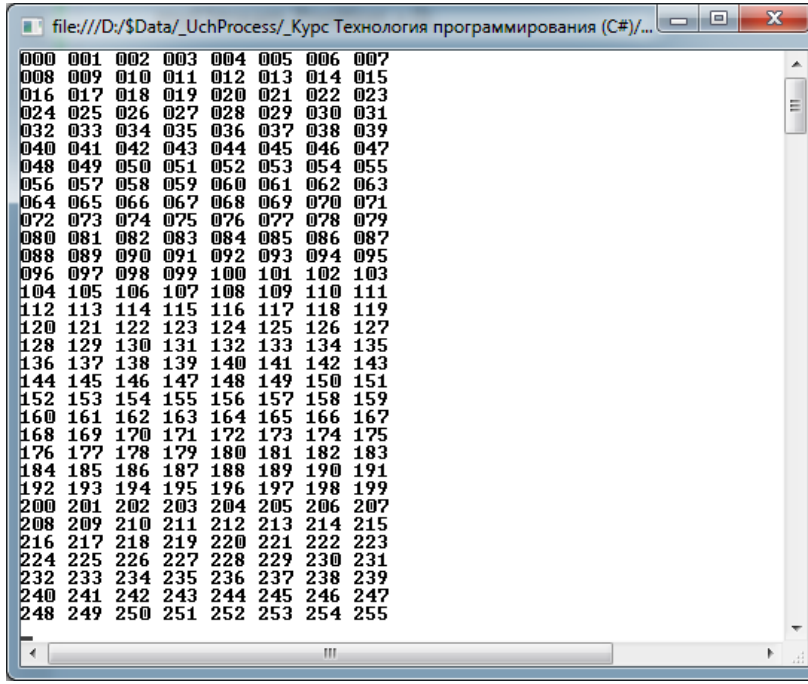

Рис.1.3 Содержимое файла *test.dat*

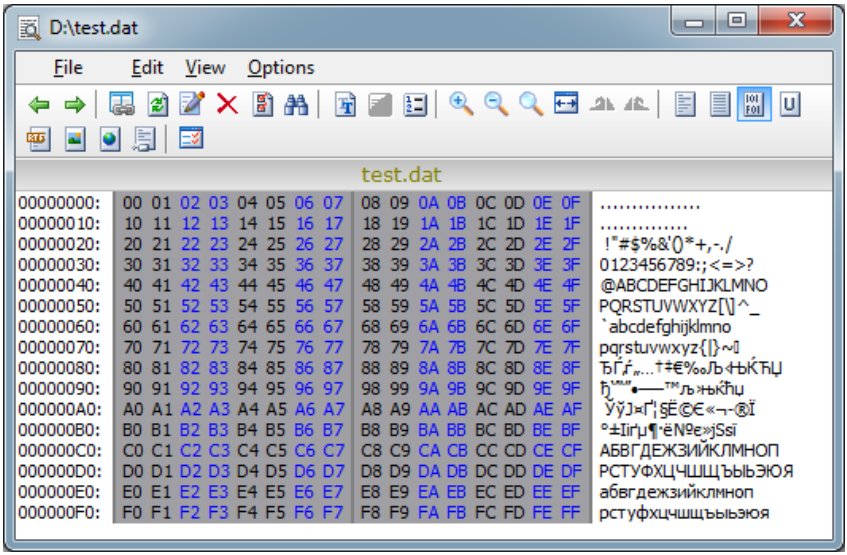

Рис.1.3. Двоичный дамп созданного файла *test.dat*

1.3. Класс *MemoryStream*

Класс *MemoryStream* во многом напоминает работу с *FileStream* с той лишь разницей, что двоичные данные размещаются не в файле на диске, а в оперативной памяти. Поскольку *FileStream* и *MemoryStream* происходят от одного базового класса – *Stream*, то многие их члены являются общими.

Кроме членов, унаследованных от *Stream*, *MemoryStream* определяет дополнительные члены, которые представлены в табл.1.2.

Разработчики библиотеки базовых классов предусмотрели возможность взаимодействия между *MemoryStream* и *FiIeStream*. С помощью метода *WriteTo*( ) можно передать данные из оперативной памяти в файл. При помощи метода *ToArray( )* можно переместить все данные из потока *MemoryStream* в массив байт.

Таблица 1.2 – Дополнительные члены класса *MemoryStream*

Член класса	Спецификация	Описание
Capacity	*public virtual int Capacity { get; set; }*	Количество байтов, выделенных под поток
GetBuffer( )	*public virtual byte[] GetBuffer( )*	Возвращает массив байт, образующий поток
ToArray( )	*public virtual byte[] ToArray( )*	Записывает содержимое потока в массив байт, независимо от свойства Position
WriteTo( )	*public virtual void WriteTo(Stream stream)*	Записывает содержимое текущего потока MemoryStream в другой поток типа, производного от Stream

Пример использования класса *MemoryStream* приводится ниже. Он повторяет пример с классом *FileStream*, но все действия выполняются в оперативной памяти, после чего данные из потока в памяти переписываются в файловый поток.

```
using System;
using System.Text;
using System.IO;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         // создаём объект потока в оперативной памяти
         MemoryStream myMemStream = new MemoryStream( );
```

```
        // записываем в поток в памяти значения в диапазоне [0,255]
        for (int i = 0; i < 256; i++)
            myMemStream.WriteByte((byte)i);

        // переписываем содержимое потока в памяти в массив байт ar
        byte[] ar = myMemStream.ToArray( );

        // выводим на консоль содержимое массива ar
        for (int i = 0; i < ar.Length; i++)
        {
            Console.Write("{0:D3} ", ar[i]);
            if ((i + 1) % 8 == 0)
                Console.WriteLine( );
        }

        // переписываем поток в памяти в создаваемый файловый поток
        myMemStream.WriteTo(new FileStream("test.dat",
                                           FileMode.Create, FileAccess.Write));
    }
  }
}
```

1.4. Класс *BufferedStream*

Класс *BufferedStream* используется для организации временного хранилища информации, которая затем будет передана в постоянное хранилище.

При работе с файлами можно обойтись без использования этого класса. Но если нам важна производительность, то лучше выполнять все действия с использованием объекта *BufferedStream*, а затем перенести данные из буфера в файл на диске за один раз. Этим способом мы уменьшим число обращений к физическому файлу на диске и выиграем во времени.

Пример программы для формирования содержимого файла, который использует буферный поток (объект *buf* класса *BufferedStream*), приведен ниже. Отметим, что, вызывая метод *Close( )* объекта *buf*, мы освобождаем и буфер, и все используемые для работы с файлом ресурсы и закрываем сам файл.

```
using System;
using System.Text;
using System.IO;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         // создаём объект файлового потока для работы с файлом
         FileStream myFileStream = new FileStream("test.dat",
                                                  FileMode.OpenOrCreate,
                                                  FileAccess.ReadWrite);

         // создаём объект буферного потока для файлового потока
         BufferedStream buf = new BufferedStream(myFileStream);

         // записываем в буферный поток значения в диапазоне [0,255]
         for (int i = 0; i < 256; i++)
            buf.WriteByte((byte)i);

         // освобождаем буфер (переписываем данные в файл) и все ресурсы
         buf.Close( );
      }
   }
}
```

2. КЛАССЫ ДЛЯ ЧТЕНИЯ И ЗАПИСИ ПРОСТЫХ ТИПОВ В ДВОИЧНОМ ФОРМАТЕ

Пространство имен *System.IO* содержит два полезных класса для работы с двоичными файлами – *BinaryReader* и *BinaryWriter*. Как показано на рис.2.1, оба эти класса происходят непосредственно от *System.Object*.

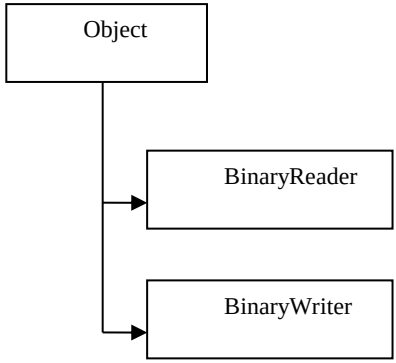

Рис.2.1. Иерархия классов

Эти типы позволяют считывать и записывать данные простых типов в поток в двоичном формате (т.е. в том виде как, они хранятся в памяти), поддерживая операции над символами и строками в необходимой кодировке. Члены класса *BinaryReader, BinaryWriter* приведены в табл.2.1–2.2.

Рассматриваемые классы удобно использовать для организации хранения данных программ в файловой системе путём разработки файлов со специальными форматами, наиболее отвечающими решаемым задачам.

В таком подходе разрабатывается удобная для работы приложения схема размещения данных в файле (формат файла), которая обеспечивается функциями записи и чтения файла, реализуемыми методами классов *BinaryWriter* и *BinaryReader*.

Таблица 2.1 – Члены класса *BinaryReader*

Член класса	Спецификация	Описание
конструктор	*BinaryReader(Stream input)* *BinaryReader(Stream input, Encoding encoding)*	Используется для инициализации объекта при его создании оператором new. *input* – определяет базовый поток ввода, *encoding* – определяет кодировку символов, которую нужно использовать при чтении из файла
BaseStream	*Stream BaseStream { get; }*	Возвращает базовый поток ввода
Close( )	*void Close( )*	Закрывает текущий поток чтения и связанный с ним базовый поток *Stream*
PeekChar( )	*int PeekChar( )*	Возвращает следующий доступный для чтения символ, не изменяя позиции чтения потока или значение –1, если в потоке нет символов
Read( )	*int Read( )*	Возвращает следующий символ из потока ввода или значение –1 , если в потоке нет символов
	int Read(byte[] buffer, int index, int count)	Считывает последовательность байт из потока, изменяя позицию чтения\записи. Если возвращаемое значение функции не ноль, то массив buffer в диапазоне индексов [*index, index* + count]содержит считанную последовательность. Возвращает число считанных байт
	int Read(char[] buffer, int index, int count)	Считывает последовательность символов из потока, изменяя позицию чтения\записи. Если возвращаемое значение функции не ноль, то массив buffer в диапазоне индексов [*index, index* + count] содержит считанную последовательность. Возвращает число считанных символов
ReadBoolean( )	*bool ReadBoolean( )*	Возвращает считанное из потока значение bool, изменяя позицию чтения на один байт вперед
ReadByte( )	*byte ReadByte( )*	Возвращает считанный байт из потока, изменяя позицию чтения на один байт вперед

Продолжение табл. 2.1

ReadBytes( )	*byte[] ReadBytes(int count)*	Возвращает считанный массив байт из потока, изменяя позицию чтения на считанное количество байт
ReadChar( )	*char ReadChar( )*	Возвращает считанный из потока символ, изменяя позицию чтения в соответствии с используемым значением Encoding и считанным символом
ReadChars( )	*char[] ReadChars(int count)*	Возвращает считанный из потока массив символов, изменяя позицию чтения в соответствии с используемым значением Encoding и количеством считанных символов
ReadDecimal( )	*decimal ReadDecimal( )*	Возвращает считанное из потока значение *decimal*, изменяя позицию чтения на 16 байт вперед
ReadDouble( )	*double ReadDouble( )*	Возвращает считанное из потока значение *double*, изменяя позицию чтения на 8 байт вперед
ReadInt16( )	*short ReadInt16( )*	Возвращает считанное из потока значение *short*, изменяя позицию чтения на 2 байт вперед
ReadInt32( )	*int ReadInt32( )*	Возвращает считанное из потока значение *int*, изменяя позицию чтения на 4 байт
ReadInt64( )	*long ReadInt64( )*	Возвращает считанное из потока значение *long*, изменяя позицию чтения на 8 байт вперед
ReadSByte( )	*sbyte ReadSByte( )*	Возвращает считанный из потока байт со знаком, изменяя позицию чтения на один байт вперед.
ReadSingle( )	*float ReadSingle( )*	Возвращает считанное из потока значение *float*, изменяя позицию чтения на 4 байт вперед
ReadString( )	*string ReadString( )*	Возвращает считаную из потока строку. Строка предваряется значением длины строки, которое закодировано как целое число блоками по 7 семь бит
ReadUInt16( )	*ushort ReadUInt16( )*	Возвращает считанное из потока значение *ushort*, изменяя позицию чтения на 2 байт вперед
ReadUInt64( )	*ulong ReadUInt64( )*	Возвращает считанное из потока значение *ulong*, изменяя позицию чтения на 8 байт вперед

Поскольку запись данных в файл осуществляется в двоичном формате, при обратной операции – чтении не возникает необходимости преобразования из символьного представления в двоичную форму хранения данных в памяти, что ускоряет работу и повышает надежность работы приложения.

Таблица 2.2 – Члены класса *BinaryWriter*

Член класса	Спецификация	Описание
конструктор	*BinaryWriter(Stream output)* *BinaryWriter(Stream output, Encoding encoding)*	Используется для инициализации объекта при его создании оператором new. *output* – определяет базовый поток вывода, *encoding* – определяет кодировку символов, которую нужно использовать при записи в файл
BaseStream	*Stream BaseStream { get; }*	Возвращает базовый поток вывода.
Close( )	*void Close( )*	Закрывает текущий поток записи и связанный с ним базовый поток *Stream*
void Flush( )	*void Flush( )*	Очищает все буферы текущего объекта записи и вызывает немедленную запись всех буферизованных данных на базовое устройство
Seek( )	*long Seek(int offset, SeekOrigin origin)*	Задает позицию в текущем потоке
WriteBoolean( )	*voi d WriteBoolean(bool value)*	Записывает в поток значение *value* длиной 1 байт, изменяя позицию записи. При этом 0 соответствует false, а 1 - true
WriteByte( )	*void WriteByte(byte value)*	Возвращает считанный байт из потока, изменяя позицию записи на 1 байт вперед
WriteBytes( )	*void WriteBytes(byte[] buffer)*	Выполняет запись массива байт в базовый поток

Продолжение табл. 2.2

Write( )	*void Write(char ch)*	Выполняет запись символа в поток и перемещает текущую позицию в потоке вперед в соответствии с используемым объектом Encoding
	void Write(char[] chars)	Выполняет запись массива символов в поток и перемещает текущую позицию в потоке в соответствии с используемым объектом Encoding и количеством записанных в поток символов
	void Write(decimal value)	Записывает *value* в текущий поток и перемещает позицию в потоке вперед на 16 байт
Write( )	*void Write(double value)*	Записывает число value в текущий поток и перемещает позицию в потоке вперед на 8 байт.
	void Write(short value)	Записывает число value в текущий поток и перемещает позицию в потоке вперед на 2 байт
	void Write(int value)	Записывает число value в текущий поток и перемещает позицию в потоке вперед на 4 байт
	void Write(long value)	Записывает число value в текущий поток и перемещает позицию в потоке вперед на 8 байт
	void Write(sbyte value)	Записывает число *value* (байт со знаком) в текущий поток и перемещает позицию в потоке вперед на 1 байт
	void Write(float value)	Записывает число value в текущий поток и перемещает позицию в потоке вперед на 4 байт
	void Write(string value)	Записывает в поток строку, предваряемую ее длиной и перемещает позицию в потоке вперед в соответствии с используемой кодировкой и количеством записанных в поток символов
	void Write(ushort value)	Записывает число *value* (число без знака) в текущий поток и перемещает позицию в потоке вперед на 2 байт
	void Write(uint value)	Записывает число *value* (число без знака) в текущий поток и перемещает позицию в потоке вперед на 4 байт

Окончание табл. 2.2

	void Write(ulong value)	Записывает число *value* (число без знака) в текущий поток и перемещает позицию в потоке вперед на 8 байт
	void Write(byte[] buffer, int index, int count)	Выполняет запись диапазона [index, index + count] массива байт *buffer* в текущий поток, изменяя позицию в потоке
	void Write(char[] chars, int index, int count)	Выполняет запись диапазона [index, index + count] массива символов *chars* в текущий поток и изменяет текущую позицию в потоке в соответствии с используемой Encoding и количеством записанных символов

Далее приводится пример, иллюстрирующий возможности этих классов.

```
using System;
using System.Text;
using System.IO;

namespace ExampleBinaryReaderWriter
{
   class Program
   {
      // класс объекта для чтения\записи из файла
      class ObjectType
      {
         public int field_Int;
         public float field_Float;
         public bool field_Bool;
         public char[] field_CharArray;

         // метод инициализации объекта
         public void InitObject( )
         {
```

```
        field_Int = 99;
        field_Float = 9984.8234f;
        field_Bool = false;
        field_CharArray = new char[] { 'H', 'e', 'l', 'l', 'o' };
    }

    // метод для текстового представления объекта
    public override  string ToString( )
    {
        string object_text = "Object state:\n";
        object_text += field_Int.ToString( ) + "\n";
        object_text += field_Float.ToString( ) + "\n";
        object_text += field_Bool.ToString( ) + "\n";
        object_text += new string(field_CharArray)+"\n";
        return object_text;
    }
}

// класс да чтения\записи объекта ObjectType из файла
class BinarySaver
{
    FileStream          fileStream;
    BinaryWriter        writer;
    BinaryReader        reader;

    // конструктор
    public BinarySaver(string filename, Encoding encode)
    {
        fileStream = new FileStream(filename,
                                    FileMode.OpenOrCreate,
                                    FileAccess.ReadWrite);

        writer = new BinaryWriter(fileStream);
```

```
    reader = new BinaryReader(fileStream);
  }

 // метод для записи в файл
  public void Write(ObjectType ob)
  {
    writer.Write(ob.field_Int);
    writer.Write(ob.field_Float);
    writer.Write(ob.field_Bool);
    writer.Write(ob.field_CharArray);
  }

  // метод для чтения из файла
  public ObjectType Read( )
  {
    ObjectType ob = new ObjectType( );

    ob.field_Int=reader.ReadInt32( );
    ob.field_Float=reader.ReadSingle( );
    ob.field_Bool=reader.ReadBoolean( );
    ob.field_CharArray=reader.ReadChars(5);

    return ob;
  }

  // метод получения дампа файла
  public void FileDump( )
  {
    reader.BaseStream.Position = 0;
    while (reader.PeekChar( ) != -1)
      Console.Write("{0:X2} ", reader.ReadByte( ));
    Console.WriteLine( );
  }
```

```
        // свойство позиция в потоке
         public long Position
        {
            set { reader.BaseStream.Position = value; }
            get { return reader.BaseStream.Position; }
        }

        // метод для закрытия потоков
        public void Close( )
        {
            if(reader!=null)
                reader.Close( );
            if(writer!=null)
                reader.Close( );
        }
    }

    static void Main(string[] args)
    {
      // создаем и инициализируем первоначальный объект до записи в файл
        ObjectType object_before_write = new ObjectType( );
        object_before_write.InitObject( );

        // создаем объект BinarySaver
        string filename = "binary.dat";
        BinarySaver saver = new BinarySaver(filename, new UTF8Encoding( )
);

        Console.WriteLine("Состояние до записи в файл:\n\n {0}\n",
                                        object_before_write.ToString( ));
```

```
        // записываем объект в файл
        Console.WriteLine("Запись объекта в файле {0}\n\n", filename);
        saver.Write(object_before_write);

        // читаем объект из файл
        Console.WriteLine("Чтение объекта из файла {0}\n\n", filename);

        saver.Position = 0;
        ObjectType object_after_read = saver.Read( );

        Console.WriteLine("Состояние после чтения из файла:\n\n {0}\n",
                                        object_after_read.ToString( ));

        // выводим дамп файла
        Console.WriteLine("Дамп файла состояния объекта:\n\n");
        saver.FileDump( );

        // закрываем потоки
        saver.Close( );
      }
   }
}
```

В примере для записи и чтения объекта программы из файла используется класс *BinarySaver*. Данные, сохраняемые в файле, представлены объектом *object_before_write*, считанные из файла – объектом *object_after_read* типа *ObjectType*. Результат работы программы представлен на рис.2.1.

Зная, какое количество байт используется для представления данных, можно нарисовать схему размещения данных в файле (рис.2.2).

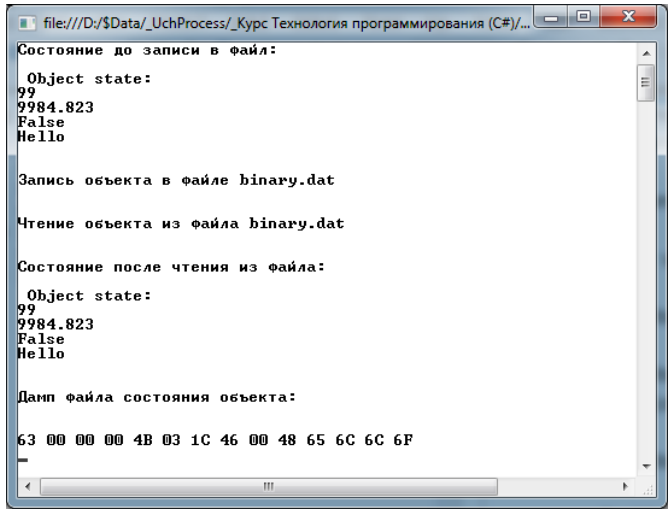

Рис.2.1. Окно выполнения программы

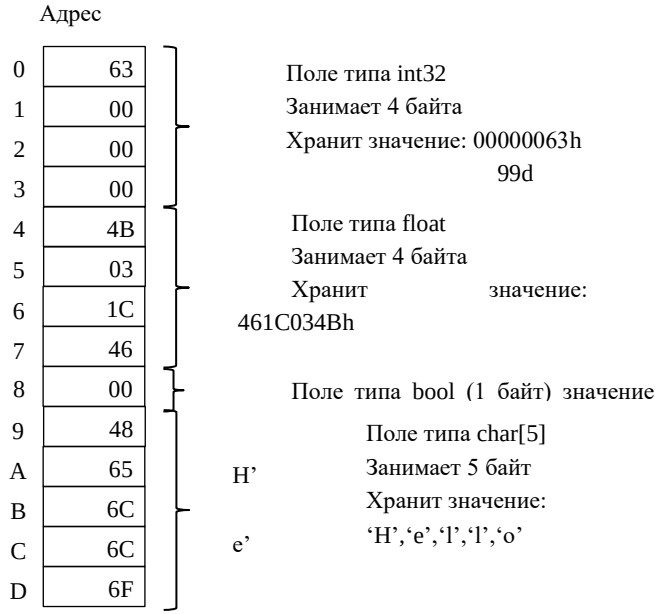

Рис.2.2. Размещение объекта *ObjectType* в файле

3. СЕРИАЛИЗАЦИЯ ДАННЫХ

При разработке программ часто возникает необходимость сохранения различных объектов на внешних носителях памяти, например, в файлах. Для решения проблем хранения данных можно следовать описанному выше способу, разрабатывая специальные форматы двоичного представления данных. Если объем таких данных очень велик (например, изображения, фильм, звук), то такой путь оправдан и эффективен.

В противном случае реализация этих задач требует значительных усилий и времени, а эффект незначительный. Для таких случаев, когда имеется потребность во внешнем хранении данных, но разрабатывать специальный формат нежелательно, можно воспользоваться сериализацией.

В C# имеется механизм, предоставляющий специальные классы для решения проблемы сохранения данных в файлах. В нём процесс сохранения объекта в файл называют сериализацией, а обратное к нему действие – десериализацией.

В терминах .NET сериализация (serialization) – это термин, описывающий процесс преобразования объекта в линейную последовательность байтов.

Обратный процесс, когда из потока байтов, содержащего всю необходимую информацию, объект восстанавливается в исходном виде, называется десериализацией (deserialization).

Службы сериализации в .NET – это весьма сложные программные модули. Они обеспечивают многие неочевидные вещи: например, когда объект сериализуется в поток, информация о всех других объектах, на которые он ссылается, также должна сериализоваться. После того как набор объектов сохранен в поток, мы можем обходиться с полученным набором байтов так, как нам захочется (куда-нибудь переслать, восстановить и использовать).

Информация о сериализации сохраняется в метаописании классов. Для выполнения сериализации объекта, каждый класс, который будет участвовать в сериализации, должен обладать атрибутом *[Serializable]*.

Если какие-либо переменные класса должны быть исключены из сериализации, достаточно просто пометить атрибутом *[NonSerialized]*. Обычно

так помечаются данные класса, которые сохранять не нужно или бессмысленно.

Для сериализации возможно использование одного из двух форматов, каждый из которых поддерживается соответствующим «форматтером» (реализует преобразование объекта из памяти в файл и наоборот).

Независимо от типа “форматтера”, он включает методы:

Deserialize( ) – десериализует поток байтов в объект;

Serialize( ) – сериализует объект в поток.

Общая схема прямого и обратного процесса представлена на рис.3.1.

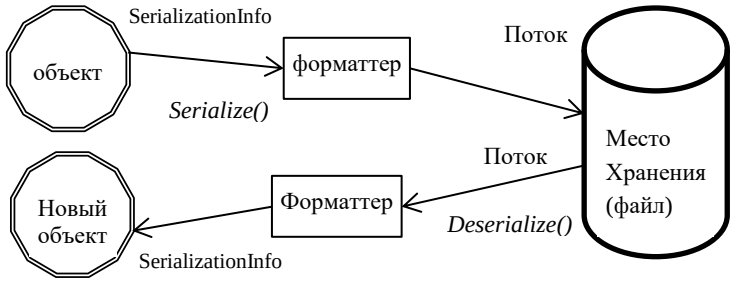

Рис.3.1. Прямой и обратный процесс сериализации

3.1. Класс *BinaryFormatter*

BinaryFormatter сериализует объекты в компактном двоичном потоке. Для его использования необходимо подключать пространство *System.Runtime.Serialization.Formatters.Binary.*

В примере процесс сериализации показан на объекте класса *Container*, содержащий внутри объект *cont* класса *ArrayList* и переменную *count*, кото-

рая отмечена как не подлежащая сериализации. При создании сериализуемый объект *myCont* инициализируется набором объектов разных типов:

1, 2.5, "Строка текста. String of text", { 1, 2, 3, 4, 5 }, 'a', "end".

Успешный результат работы примера показан на рис.3.2, а содержимое файла *binSerial.dat* – на рис.3.3.

```
using System;
using System.Text;
using System.Collections;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerialization
{

   [Serializable]
   class Container
   {
      private ArrayList   cont;

      [NonSerialized]
      private int count = 9;

      public Container(params object[] vars)
      {
         cont = new ArrayList( );
         foreach (object el in vars)
            cont.Add(el);
      }
```

```
    public void View( )
    {
        Console.WriteLine("[NonSerialized] private int count={0}\n", count);
        for (int i = 0; i < cont.Count; i++)
        {
            if (cont[i].GetType( ) != typeof(int[]))
                Console.WriteLine("{0}", cont[i].ToString( ));
            else
            {
                int[] arr = (int[])(cont[i]);

                foreach (int el in arr)
                    Console.Write("{0} ", el);
                Console.WriteLine( );
            }
        }
    }
}


class Program
{
    public static void Save( string filename, object ob)
    {
        FileStream myStream = File.Create(filename);

        BinaryFormatter binFormatter = new BinaryFormatter( );

        binFormatter.Serialize(myStream, ob);

        myStream.Close( );
    }
```

```
    public static object Restore(string filename)
    {
        FileStream myStream = File.OpenRead(filename);
        BinaryFormatter binFormatter = new BinaryFormatter( );

        return binFormatter.Deserialize(myStream);
    }

    static void Main(string[] args)
    {
        string filename = "binSerial.dat";

        Container myCont = new Container(1, 2.5,
                                         "Строка текста. String of text",
                                         new int[] { 1, 2, 3, 4, 5 }, 'a',
                                        "end");

        Console.WriteLine("Состояние объекта до сериализации:\n");
        myCont.View( );

        // сериализация
        Save(filename,  myCont);

        // десериализация
        myCont = (Container)Restore(filename);

        Console.WriteLine("Состояние объекта после десериализации:\n");
        myCont.View( );
    }
  }
}
```

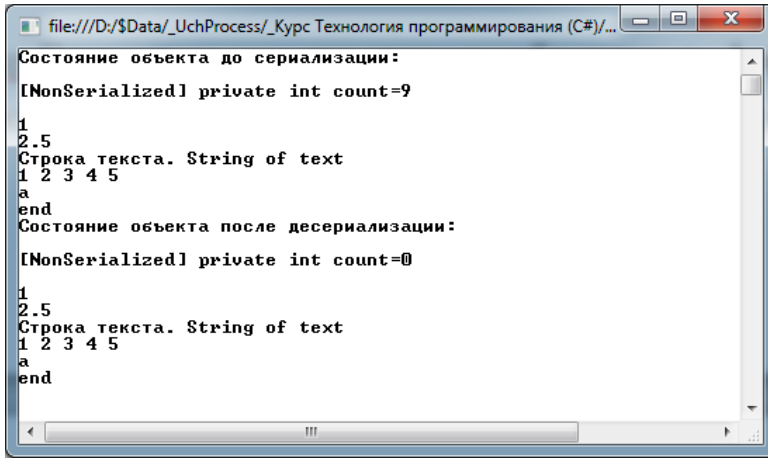

Рис.3.2. Результат работы программы

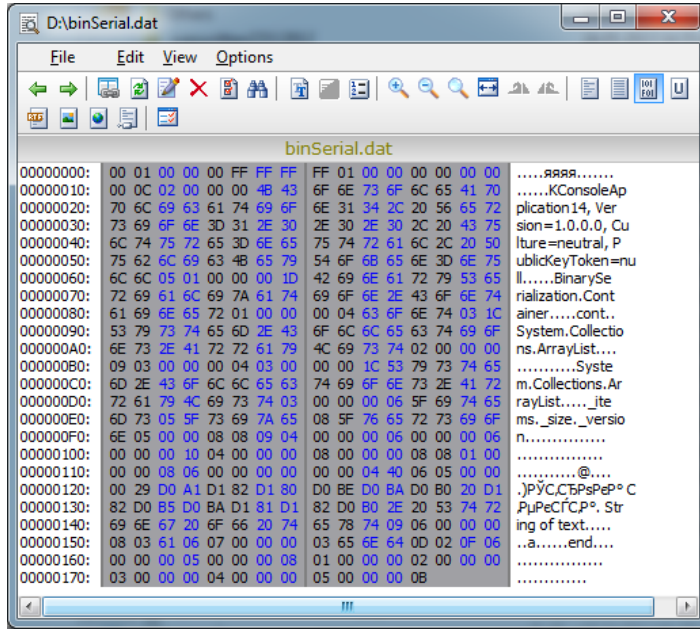

Рис.3.3. Содержимое файла *binSerial.dat*

3.2. Класс *SoapFormatter*

Класс *SoapFormatter* – сериализует объекты в форме сообщения протокола SOAP (Simple Object Access Protocol – простого протокола доступа к объектам) в формате XML. Для использования этого класса необходимо в проект добавить ссылку на модуль *System.Runtime.Serialization.Formatters.Soap* и подключить одноименное пространство имён.

Пример использования этого класса приводится ниже. Если сравнить две рассмотренные реализации, то в тексте, кроме разных объектов сериализации, они ничем не отличаются. Отличается только файл сериализации *soapSerial.dat*, он представлен на рис.3.4.

```
using System;
using System.Text;
using System.Collections;
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;

namespace SoapSerialization
{
   [Serializable]
   class Container
   {
      private ArrayList cont;

      [NonSerialized]
      private int count = 9;

      public Container(params object[] vars)
      {
         cont = new ArrayList();
         foreach (object el in vars)
            cont.Add(el);
```

```
    }

    public void View()
    {
        Console.WriteLine("[NonSerialized] private int count={0}\n", count);
        for (int i = 0; i < cont.Count; i++)
        {
            if (cont[i].GetType() != typeof(int[]))
                Console.WriteLine("{0}", cont[i].ToString());
            else
            {
                int[] arr = (int[])(cont[i]);

                foreach (int el in arr)
                    Console.Write("{0} ", el);
                Console.WriteLine();
            }
        }
    }
}

class Program
{
    public static void Save(string filename, object ob)
    {
        FileStream myStream = File.Create(filename);

        SoapFormatter soapFormatter = new SoapFormatter();

        soapFormatter.Serialize(myStream, ob);

        myStream.Close();
    }
```

```
    public static object Restore(string filename)
    {
        FileStream myStream = File.OpenRead(filename);

        SoapFormatter soapFormatter = new SoapFormatter();

        return soapFormatter.Deserialize(myStream);
    }
    static void Main(string[] args)
    {
        Console.BackgroundColor = ConsoleColor.White;
        Console.ForegroundColor = ConsoleColor.Black;
        Console.Clear();

        string filename = "soapSerial.dat";

        Container myCont = new Container(1, 2.5,
                                         "Строка текста. String of text",
                                         new int[] { 1, 2, 3, 4, 5 },
                                         'a', "end");
        Console.WriteLine("Состояние объекта до сериализации:\n");
        myCont.View();
        // сериализация
        Save(filename, myCont);

        // десериализация
        myCont = (Container)Restore(filename);

        Console.WriteLine("Состояние объекта после десериализации:\n");
        myCont.View();
    }
  }
}
```

Если вас не устроит ни один из рассмотренных классов, вы можете создать свой собственный формат сериализации и соответствующий ему класс «форматтера». Для этой цели необходимо использовать классы пространства *System.Runtime.Seria1ization*.

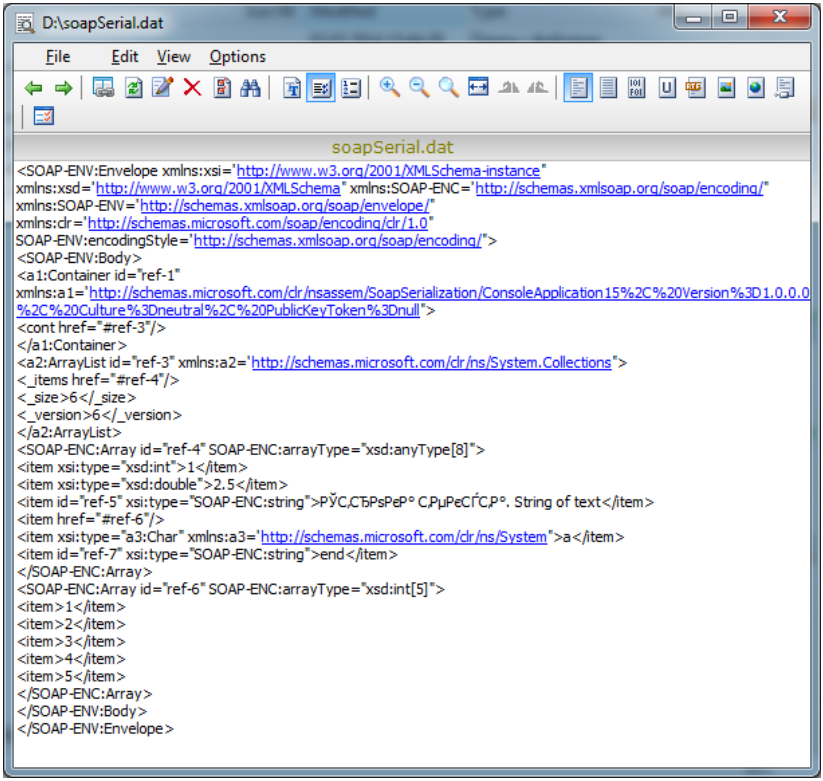

Рис.3.4 Содержимое файла *soapSerial.dat*

4. КОНТРОЛЬНЫЕ ЗАДАНИЯ

Цель работы – изучение и освоение возможностей языка C# и платформы .Net Framework для работы с двоичными файлами и потоками.

Для выполнения работ могут быть использованы системы программирования MS Visual Studio .Net 2010, 2012, 2013 MS Visual C# Express Edition 2010, 2012, 2013.

Задание

1) Изучить иерархию классов для работы с двоичными файлами и потоками и функциональные возможности классов: *Stream, FileStream, MemoryStream, BufferedStream.*

2) Изучить функциональные возможности классов для чтения\записи простых типов в двоичные потоки: *BinaryReader, BinaryWriter*.

3) Освоить функциональные возможности и приемы работы классов сериализации данных: *BinaryFormatter, SoapFormatter*.

4) Освоить способы сохранения данных программы с использованием двоичных файлов.

5) Решить задачу на закрепление полученных знаний:

Разработать консольное приложение для ввода и редактирования описания однотипных объектов (тип объектов определяется вариантом задания, см. табл. 4.1).

Требования к программной реализации

1. Во время выполнения приложения объекты хранятся в оперативной памяти. Для хранения объектов в памяти использовать объект *MemoryStream*.

2. Когда приложение не выполняется, объекты хранятся в файлах. Использовать следующие типы файлов:

– двоичный файл;

– файл сериализации.

3. Управление приложением осуществляется пользователем посредством экранного меню, которое включает пункты:

1) *Очистить*. При выборе все сущности из памяти удаляются.

2) *Добавить объект*. Вводится описания нового объекта, добавляемое в конец списка описаний.
3) *Найти объект*. Определяется поиск объекта (определение позиции размещения его описания в памяти). Поиск может осуществляться на выбор пользователя: по указанию значения выбранного описателя объекта или по указанию индекса объекта в памяти. На экран выводится сообщение: «Объект найден» или «Объект не найден», в зависимости от результата поиска. Найденный объект считается текущим, над которым выполняются действия.
4) *Отобразить описание объекта*. На экран выводится описание текущего объекта.
5) *Редактировать описание объекта*. Выполняется редактирование текущего объекта пользователем – для описателей объекта по выбору пользователя выполняется запрос нового значения, вносятся изменения в описание объекта в памяти.
6) *Сохранить объекты*. Описание объектов из памяти переписывается в файл хранения (имя файла вводится пользователем).
7) *Загрузить объекты*. Описание объектов из файла (имя файла вводится пользователем) переписывается в память.
8) *Сериализовать.* Описание объектов в памяти сериализуется в файл (на выбор пользователя binary или soap).
9) *Десериализовать.* Описание объектов из файла (указывается пользователем) десериализуеися в память.
10) *Обработать*. Вызывается метод обработки списка описания объектов (определяется вариантом, см.табл.4.1).
11) *Листинг.* Описание объектов выводится в текстовой файл (имя файла вводится пользователем).

По результатам выполнения работы студент должен подготовить и представить преподавателю отчет, который включает: постановку задачи, описание метода решения, описание разработанных классов (структура данных и алгоритмы), описание формата хранения объектов в памяти и в файле, текст и описание программы, тестовый пример, выводы.

Таблица 4.1 – Варианты задания

№	Объект и его описатели	Задача для п.10 меню
1	*Библиотека* (авторы, название, изд-во, год издания, кол-во страниц, кол-во книг, состояние)	Сформировать список книг с годом издания старше заданного в плохом состоянии.
2	*Ведомость успеваемости группы* (ФИО, Оценка_1, Оценка_2, Оценка_3, Оценка_4, Оценка_5, зачет_1, зачет_2, зачет_3, зачет_4, зачет_5)	Сформировать список студентов, которых необходимо отчислить за неуспеваемость.
3	*Коллекция предметов* (название, год, вес, стоимость, автор)	Определить К самых дорогих экспонатов коллекции.
4	*Изделия* (название, вес, стоимость, % метала, % износа, год ремонта)	Определить список изделий с износом большим 60%, и вес металла, после их утилизации.
5	*Плитка* (название, размер1, размер2, цвет, мат\глянц, цена 1 $м^2$, страна, налич $м^2$, код плитки)	Задается число в $м^2$. Определить список видов плитки, наличие которой на складе меньше заданного.
6	*Приборы* (название, код, вес, % серебра, % золота, год)	Задан год. Для приборов с годом старше заданного определить абсолютное содержание серебра и золота.
7	*Вклады* (ФИО, дата вклада, годовой %, сумма, срок вклада, состояние (действует, закрыт))	Определить сумму выплат банком по всем вкладам, срок истечения которых совпадает с заданной датой.

Продолжение табл. 4.1

8	*Кредиты* (ФИО, дата выдачи, годовой %, сумма, срок кредита, состояние (действует, закрыт)	Определить сумму возврата банком по кредитам, которые истекают на эту дату.
9	*Погода* (день, месяц, год, температура, состояние (дождь, снег, ясно, облачно))	Заданы месяц, год1 и год2. Определить, насколько изменилась среднемесячная температура в году (год2) в заданном месяце по сравнению с годом (год1).
10	*Овощи* (название, код партии, вес, цена за кг, остаток_вес, допустимый срок хранения в днях, дата поступления)	Задается дата1. Определить общие потери всем овощам в грн, для которых истек срок хранения.
11	*Конфеты* (название, фирма производитель, цена за кг, вид (шоколадн, карамель), наличие кг, кол-во конфет на 1 кг)	Вводится натуральное число К. Какой должен быть вес подарка, если в подарок включить по К конфет каждого названия, и какая его будет стоимость?
12	*Квартиры дома* (ФИО владельца, кол-во человек, наличие хол.воды, наличие гор.воды, площадь квартиры)	Норма на чел.: хол. воды – 8 куб., гор. воды – 3 куб. Известны стоимость хол. и подогрева гор. воды за куб. Какая сумма платы дома за воду в мес?
13	*Материалы* (название, наличие кг, цена за кг, код материала)	Определить среднюю стоимость материалов за кг.
14	*Краска* (название, тип (масляная, нитро, водоэмульс), цвет, цена за кг, в наличии кг, расход на кв.м)	Задан тип краски, площадь. Определить, сколько разных красок в наличии для выполнения покраски данной площади.
15	*Журналы* (код, назавание, месяц, год, кол-во стр, вес, кол-во шт.)	Посчитать, сколько можно сдать кг макулатуры, если списать все журналы за заданный год.
16	*Валюта* (название, курс в грн. на начало года, курс в грн, на конец года, % по вкладу)	Задана сумма в грн. Для каждой валюты рассчитать доход в грн, за год, если сумму вложить в этой валюте.

СПИСОК ЛИТЕРАТУРЫ

1. Рейли Д. Создание приложений Microsoft ASP.Net / Д.Рейли : пер.с англ.– М.: Изд.-торгов.дом. «Русская редакция», 2002. – 480с., ил.

2. Петцольд Ч. Программирование для Microsoft Windows на C#: В 2-х т. Т.1. Ч. Петцольд : пер. с англ. – М.: Изд.-торгов.дом «Русская Редакция», 2002. – 576 с., ил.

3. Петцольд Ч. Программирование для Microsoft Windows на C#: В 2-х т. Т.2. Ч. Петцольд : пер. с англ. – М.: Изд.-торгов.дом «Русская Редакция», 2002.– 624 с., ил.

4. Лабор В. В. Си Шарп: Создание приложений для Windows / В. В. Лабор.– Мн.: Харвест, 2003. – 384 с.

5. Рихтер Дж. Программирование на платформе Microsoft .NET Framework / Дж. Рихтер : пер. с англ. – 2-е изд., испр. – М.: Изд.-торгов.дом «Русская Редакция», 2003. – 512 стр., ил.

6. Разработка Windows-приложений на Microsoft Visual Basic .NET и Microsoft Visual C# -NET: Учеб.курс MCAD/MCSD : пер. с англ. – М.: Изд.-торгов.дом «Русская Редакция», 2003. – 512 стр., ил.

7. Троелсен Э. C# и платформа NET. Библиотека программиста / Э. Троелсен. – Спб.: Питер, 2003. – 800с.,ил.

8. Анализ требований и создание архитектуры решений на основе Microsoft .NET: Учеб.курс MCSD: пер. с англ.– М.: Изд.-торгов.дом “Русская Редакция”, 2004. – 416 стр., ил.

9. Шилдт Г. Полный справочник по C# / Шилдт Г. : пер. с англ. – М.: Изд.дом «Вильямс», 2004. – 752 с., ил.

10. Бишоп Дж. C# в кратком изложении / Дж. Бишоп, Н. Хорспул : пер. с англ. – М.: «Бином», Лаборатория знаний, 2005. – 472с., ил.

Содержание

Навчальне видання

Методичні вказівки
до лабораторної роботи за темою
«Робота з двійковими файлами в мові C#» з дисципліни
«Технології програмування»
для студентів спеціальностей
122 – Комп'ютерні науки та інформаційні технології,
124 – Системний аналіз

Російською мовою

Укладачі: МАЛИХ Олег Миколайович
КОЖИН Юрій Миколайович
ПРОКОПЕНКОВ Володимир Пилипович

Відповідальний за випуск *О.С. Куценко*
Роботу до друку рекомендував *О.В. Горілий*

Редактор *О.С. Самініна*

План 2016 , поз. 90

Підп. до друку 23.03.2017 Формат 60x84 1/16 Папір офсетний.
Riso-друк. Гарнітура Таймс. Ум.друк.арк. 1,7
Наклад 25 прим. Зам. № Ціна договірна.

Видавничий центр НТУ "ХПІ",
вул. Кирпичова, 21, м.Харків-2, 61002
Свідоцтво суб'єкта видавничої справи ДК №3657 від 24.12.2009 р.

ООО Планета Прінт
40