

Министерство образования Республики Беларусь
«ПОЛОЦКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ. ЕВФРОСИНИИ
ПОЛОЦКОЙ»

Факультет информационных технологий
Кафедра технологий программирования

**Методические указания для выполнения
лабораторной работы №10
по курсу «Конструирование программного
обеспечения»**

«Работа с файлами в программах на языке высокого уровня»

Полоцк, 2022 г.

ЦЕЛЬ РАБОТЫ

Познакомится с работой с файлами. Разобрать методы для работы с файлами в C++ и C#. На основе примеров, приведенных в данной лабораторной работе, выполнить свой вариант практического задания.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Работа с файлами в C#

Классы File и FileInfo

Для работы с файлами предназначена пара классов File и FileInfo. С их помощью мы можем создавать, удалять, перемещать файлы, получать их свойства и многое другое.

FileInfo

Некоторые полезные методы и свойства класса FileInfo:

- CopyTo(path): копирует файл в новое место по указанному пути path
- Create(): создает файл
- Delete(): удаляет файл
- MoveTo(destFileName): перемещает файл в новое место
- Свойство Directory: получает родительский каталог в виде объекта DirectoryInfo
- Свойство DirectoryName: получает полный путь к родительскому каталогу
- Свойство Exists: указывает, существует ли файл
- Свойство Length: получает размер файла
- Свойство Extension: получает расширение файла
- Свойство Name: получает имя файла
- Свойство FullName: получает полное имя файла

Для создания объекта FileInfo применяется конструктор, который получает в качестве параметра путь к файлу:

```
FileInfo fileInf = new FileInfo(@"C:\app\content.txt");
```

File

Класс File реализует похожую функциональность с помощью статических методов:

- Copy(): копирует файл в новое место
- Create(): создает файл
- Delete(): удаляет файл
- Move: перемещает файл в новое место
- Exists(file): определяет, существует ли файл

Пути к файлам

Для работы с файлами можно применять как абсолютные, так и относительные пути:

```
// абсолютные пути
string path1 = @"C:\Users\eugene\Documents\content.txt"; // для
Windows
string path2 = "C:\\Users\\eugene\\Documents\\content.txt"; // для
Windows
string path3 = "/Users/eugene/Documents/content.txt"; // для
MacOS/Linux

// относительные пути
string path4 = "MyDir\\content.txt"; // для Windows
string path5 = "MyDir/content.txt"; // для MacOS/Linux
```

Получение информации о файле

```
string path = @"C:\Users\eugene\Documents\content.txt";
// string path = "/Users/eugene/Documents/content.txt"; // для
MacOS/Linux
FileInfo fileInfo = new FileInfo(path);
if (fileInfo.Exists)
{
    Console.WriteLine($"Имя файла: {fileInfo.Name}");
    Console.WriteLine($"Время создания: {fileInfo.CreationTime}");
    Console.WriteLine($"Размер: {fileInfo.Length}");
}
```

Удаление файла

```
string path = @"C:\app\content.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    fileInf.Delete();
    // альтернатива с помощью класса File
    // File.Delete(path);
}
```

Перемещение файла

```
string path = @"C:\OldDir\content.txt";
string newPath = @"C:\NewDir\index.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    fileInf.MoveTo(newPath);
    // альтернатива с помощью класса File
    // File.Move(path, newPath);
}
```

Если файл по новому пути уже существует, то с помощью дополнительного параметра можно указать, надо ли перезаписать файл (при значении true файл перезаписывается)

```
string path = @"C:\OldDir\content.txt";
string newPath = @"C:\NewDir\index.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    fileInf.MoveTo(newPath, true);
    // альтернатива с помощью класса File
    // File.Move(path, newPath, true);
}
```

Копирование файла

```
string path = @"C:\OldDir\content.txt";
string newPath = @"C:\NewDir\index2.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    fileInf.CopyTo(newPath, true);
    // альтернатива с помощью класса File
    // File.Copy(path, newPath, true);
}
```

Метод `CopyTo` класса `FileInfo` принимает два параметра: путь, по которому файл будет копироваться, и булево значение, которое указывает, надо ли при копировании перезаписывать файл (если `true`, как в случае выше, файл при копировании перезаписывается). Если же в качестве последнего параметра передать значение `false`, то если такой файл уже существует, приложение выдаст ошибку.

Метод `Copy` класса `File` принимает три параметра: путь к исходному файлу, путь, по которому файл будет копироваться, и булево значение, указывающее, будет ли файл перезаписываться.

Чтение и запись файлов

В дополнение к вышерассмотренным методам класс `File` также предоставляет ряд методов для чтения-записи текстовых и бинарных файлов:

- **`AppendAllLines(String, IEnumerable<String>) / AppendAllLinesAsync(String, IEnumerable<String>, CancellationToken)`**

добавляют в файл набор строк. Если файл не существует, то он создается

- **`AppendAllText(String, String) / AppendAllTextAsync(String, String, CancellationToken)`**

добавляют в файл строку. Если файл не существует, то он создается

- **`byte[] ReadAllBytes (string path) / Task<byte[]> ReadAllBytesAsync (string path, CancellationToken cancellationToken)`**

считывают содержимое бинарного файла в массив байтов

- **`string[] ReadAllLines (string path) / Task<string[]> ReadAllLinesAsync (string path, CancellationToken cancellationToken)`**

считывают содержимое текстового файла в массив строк

- **`string ReadAllText (string path) / Task<string> ReadAllTextAsync (string path, CancellationToken cancellationToken)`**

считывают содержимое текстового файла в строку

- **`IEnumerable<string> ReadLines (string path)`**

считывают содержимое текстового файла в коллекцию строк

- **`void WriteAllBytes (string path, byte[] bytes) / Task WriteAllBytesAsync (string path, byte[] bytes, CancellationToken cancellationToken)`**

записывают массив байт в бинарный файл. Если файл не существует, он создается. Если существует, то перезаписывается

- **void WriteAllLines (string path, string[] contents) / Task WriteAllLinesAsync (string path, IEnumerable<string> contents, CancellationToken cancellationToken)**

записывают массив строк в текстовый файл. Если файл не существует, он создается. Если существует, то перезаписывается

- **WriteAllText (string path, string? contents) / Task WriteAllTextAsync (string path, string? contents, CancellationToken cancellationToken)**

записывают строку в текстовый файл. Если файл не существует, он создается. Если существует, то перезаписывается

Как видно, эти методы покрывают практически все основные сценарии - чтение и запись текстовых и бинарных файлов. Причем в зависимости от задачи можно применять как синхронные методы, так и их асинхронные аналоги.

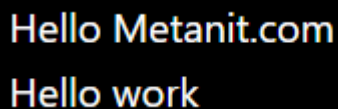
Например, запишем и считаем обратно в строку текстовый файл:

```
string path = @"c:\app\content.txt";

string originalText = "Hello Metanit.com";
// запись строки
await File.WriteAllTextAsync(path, originalText);
// дозапись в конец файла
await File.AppendAllTextAsync(path, "\nHello work");

// чтение файла
string fileText = await File.ReadAllTextAsync(path);
Console.WriteLine(fileText);
```

Консольный вывод:



```
Hello Metanit.com
Hello work
```

Стоит отметить, что при добавлении текста я добавил в строку последовательность "\n", которая выполняет перевод на следующую строку. Благодаря этому добавляемый текст располагается в файле на новой строке.

Если мы хотим, что в файле изначально шло добавление на новую строку, то для записи стоит использовать метод WriteAllLines/ WriteAllLinesAsync, а для добавления - AppendAllLines / AppendAllLinesAsync

```
await File.WriteAllLinesAsync(path, new[] { "Hello Metanit.com",
"Hello work" });
```

Аналогично при чтении файла если мы хотим каждую строку файла считать отдельно, то вместо `ReadAllText` / `ReadAllTextAsync` применяется `ReadAllLines` / `ReadAllTextAsync`.

Кодировка

В качестве дополнительного параметра методы чтения-записи текстовых файлов позволяют установить кодировку в виде объекта `System.Text.Encoding`:

```
using System.Text;

string path = "/Users/eugene/Documents/app/content.txt";

string originalText = "Привет Metanit.com";
// запись строки
await File.WriteAllTextAsync(path, originalText, Encoding.Unicode);
// дозапись в конец файла
await File.AppendAllTextAsync(path, "\nПривет мир",
Encoding.Unicode);

// чтение файла
string fileText = await File.ReadAllTextAsync(path,
Encoding.Unicode);
Console.WriteLine(fileText);
```

Для установки кодировки при записи и чтении здесь применяется встроенное значение `Encoding.Unicode`. Также можно указать название кодировки, единственное следует удостовериться, что текущая операционная система поддерживает выбранную кодировку:

```
using System.Text;

string path = @"c:\app\content.txt";

string originalText = "Hello Metanit.com";
// запись строки
await File.WriteAllTextAsync(path, originalText,
Encoding.GetEncoding("iso-8859-1"));
// дозапись в конец файла
await File.AppendAllTextAsync(path, "\nHello code",
Encoding.GetEncoding("iso-8859-1"));

// чтение файла
string fileText = await File.ReadAllTextAsync(path,
Encoding.GetEncoding("iso-8859-1"));
Console.WriteLine(fileText);
```

FileStream. Чтение и запись файла

Класс `FileStream` представляет возможности по считыванию из файла и записи в файл. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Создание `FileStream`

Для создания объекта `FileStream` можно использовать как конструкторы этого класса, так и статические методы класса `File`. Конструктор `FileStream` имеет множество перегруженных версий, из которых отмечу лишь одну, самую простую и используемую:

```
FileStream(string filename, FileMode mode)
```

Здесь в конструктор передается два параметра: путь к файлу и перечисление `FileMode`. Данное перечисление указывает на режим доступа к файлу и может принимать следующие значения:

- `Append`: если файл существует, то текст добавляется в конец файл. Если файла нет, то он создается. Файл открывается только для записи.
- `Create`: создается новый файл. Если такой файл уже существует, то он перезаписывается
- `CreateNew`: создается новый файл. Если такой файл уже существует, то приложение выбрасывает ошибку
- `Open`: открывает файл. Если файл не существует, выбрасывается исключение
- `OpenOrCreate`: если файл существует, он открывается, если нет - создается новый
- `Truncate`: если файл существует, то он перезаписывается. Файл открывается только для записи.

Другой способ создания объекта `FileStream` представляют статические методы класса `File`:

```
FileStream File.Open(string file, FileMode mode);  
FileStream File.OpenRead(string file);  
FileStream File.OpenWrite(string file);
```

Первый метод открывает файл с учетом объекта `FileMode` и возвращает файловой поток `FileStream`. У этого метода также есть несколько перегруженных версий. Второй метод открывает поток для чтения, а третий открывает поток для записи.

Заккрытие потока

Класс `FileStream` для освобождения всех реурсов, связанных с файлом, реализует интерфейс `IDisposable`. Соответственно после завершения работы с `FileStream` необходимо освободить связанный с ним файл вызовом метода `Dispose`. Для корректного закрытия можно вызвать метод `Close()`, который вызывает метод `Dispose`:

```
FileStream? fstream = null;
try
{
    fstream = new FileStream("note3.dat", FileMode.OpenOrCreate);
    // операции с fstream
}
catch (Exception ex)
{ }
finally
{
    fstream?.Close();
}
```

Либо можно использовать конструкцию `using`, которая автоматически освободит все связанные с `FileStream` ресурсы:

```
using (FileStream fstream = new FileStream("note3.dat",
FileMode.OpenOrCreate))
{
    // операции с fstream
}
```

Свойства и методы `FileStream`

Рассмотрим наиболее важные свойства класса `FileStream`:

- Свойство `Length`: возвращает длину потока в байтах
- Свойство `Position`: возвращает текущую позицию в потоке
- Свойство `Name`: возвращает абсолютный путь к файлу, открытому в `FileStream`

Для чтения/записи файлов можно применять следующие методы класса `FileStream`:

- `void CopyTo(Stream destination)`: копирует данные из текущего потока в поток `destination`

- `Task CopyToAsync(Stream destination):` асинхронная версия метода `CopyTo`

- `void Flush():` сбрасывает содержимое буфера в файл

- `Task FlushAsync():` асинхронная версия метода `Flush`

- `int Read(byte[] array, int offset, int count):` считывает данные из файла в массив байтов и возвращает количество успешно считанных байтов. Принимает три параметра:

- `array` - массив байтов, куда будут помещены считываемые из файла данные

- `offset` представляет смещение в байтах в массиве `array`, в который считанные байты будут помещены

- `count` - максимальное число байтов, предназначенных для чтения. Если в файле находится меньшее количество байтов, то все они будут считаны.

- `Task<int> ReadAsync(byte[] array, int offset, int count):` асинхронная версия метода `Read`

- `long Seek(long offset, SeekOrigin origin):` устанавливает позицию в потоке со смещением на количество байт, указанных в параметре `offset`.

- `void Write(byte[] array, int offset, int count):` записывает в файл данные из массива байтов. Принимает три параметра:

- `array` - массив байтов, откуда данные будут записываться в файл

- `offset` - смещение в байтах в массиве `array`, откуда начинается запись байтов в поток

- `count` - максимальное число байтов, предназначенных для записи

- `Task WriteAsync(byte[] array, int offset, int count):` асинхронная версия метода `Write`

Чтение и запись файлов

`FileStream` представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру, `FileStream` может быть очень даже полезен для извлечения определенных порций информации и ее обработки.

Посмотрим на примере считывания-записи в текстовый файл:

```
using System.Text;
```

```
string path = @"C:\app\note.txt";    // путь к файлу
```

```

string text = "Hello METANIT.COM"; // строка для записи

// запись в файл
using (FileStream fstream = new FileStream(path,
    FileMode.OpenOrCreate))
{
    // преобразуем строку в байты
    byte[] buffer = Encoding.Default.GetBytes(text);
    // запись массива байтов в файл
    await fstream.WriteAsync(buffer, 0, buffer.Length);
    Console.WriteLine("Текст записан в файл");
}

// чтение из файла
using (FileStream fstream = File.OpenRead(path))
{
    // выделяем массив для считывания данных из файла
    byte[] buffer = new byte[fstream.Length];
    // считываем данные
    await fstream.ReadAsync(buffer, 0, buffer.Length);
    // декодируем байты в строку
    string textFromFile = Encoding.Default.GetString(buffer);
    Console.WriteLine($"Текст из файла: {textFromFile}");
}

```

Разберем этот пример. Вначале определяем путь к файлу и текст для записи в файл.

И при чтении, и при записи для создания и удаления объекта `FileStream` используется конструкция `using`, по завершению которой у созданного объекта `FileStream` автоматически вызывается метод `Dispose`, и, таким образом, объект уничтожается.

Поскольку операции с файлами могут занимать продолжительное время и являются узким местом в работе программы, рекомендуется использовать асинхронные версии методов `FileStream`. И при записи, и при чтении применяется объект кодировки `Encoding.Default` из пространства имен `System.Text`. В данном случае мы используем два его метода: `GetBytes` для получения массива байтов из строки и `GetString` для получения строки из массива байтов.

В итоге введенная нами строка записывается в файл `note.txt`. И мы получим следующий консольный вывод:

Текст записан в файл

Текст из файла: Hello METANIT.COM

Записанный файл по сути представляет бинарный файл (не текстовый), хотя если мы в него запишем только строку, то сможем посмотреть в удобочитаемом виде этот файл, открыв его в текстовом редакторе. Однако если мы в него запишем случайные байты, например:

```
fstream.WriteByte(13);  
fstream.WriteByte(103);
```

То у нас могут возникнуть проблемы с его пониманием. Поэтому для работы непосредственно с текстовыми файлами предназначены отдельные классы - `StreamReader` и `StreamWriter`.

Произвольный доступ к файлам

Нередко бинарные файлы представляют определенную структуру. И, зная эту структуру, мы можем взять из файла нужную порцию информации или наоборот записать в определенном месте файла определенный набор байтов. Например, в wav-файлах непосредственно звуковые данные начинаются с 44 байта, а до 44 байта идут различные метаданные - количество каналов аудио, частота дискретизации и т.д.

С помощью метода `Seek()` мы можем управлять положением курсора потока, начиная с которого производится считывание или запись в файл. Этот метод принимает два параметра: `offset` (смещение) и позиция в файле. Позиция в файле описывается тремя значениями:

- `SeekOrigin.Begin`: начало файла
- `SeekOrigin.End`: конец файла
- `SeekOrigin.Current`: текущая позиция в файле

Курсор потока, с которого начинается чтение или запись, смещается вперед на значение `offset` относительно позиции, указанной в качестве второго параметра. Смещение может быть отрицательным, тогда курсор сдвигается назад, если положительное - то вперед.

Рассмотрим простой пример:

```
using System.Text;  
  
string path = "note.dat";  
  
string text = "hello world";  
  
using (FileStream fstream = new FileStream(path,  
    FileMode.OpenOrCreate))  
{  
    // преобразуем строку в байты
```

```

byte[] input = Encoding.Default.GetBytes(text);
// запись массива байтов в файл
fstream.Write(input, 0, input.Length);
Console.WriteLine("Текст записан в файл");
}
// чтение части файла
using (FileStream fstream = new FileStream(path,
FileMode.OpenOrCreate))
{
    // перемещаем указатель в конец файла, до конца файла- пять
байт
    fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца
потока

    // считываем четыре символов с текущей позиции
byte[] output = new byte[5];
await fstream.ReadAsync(output, 0, output.Length);
// декодируем байты в строку
string textFromFile = Encoding.Default.GetString(output);
Console.WriteLine($"Текст из файла: {textFromFile}"); //
world
}

```

Вначале записываем в файл текст "hello world". Затем снова обращаемся к файлу для считывания. Сначала перемещаем курсор на пять символов назад относительно конца файлового потока:

```
fstream.Seek(-5, SeekOrigin.End)
```



То есть после выполнения этого вызова курсор будет стоять на позиции символа "w".

После этого считываем пять байт начиная с символа "w". В кодировке по умолчанию 1 символ будет представлять 1 байт. Поэтому чтение 5 байт будет эквивалентно чтению пяти символов: "world".

Соответственно мы получим следующий консольный вывод:

```

Текст записан в файл
Текст из файла: world

```

Рассмотрим чуть более сложный пример - с записью начиная с некоторой позиции:

```
using System.Text;
string path = "note2.dat";
string text = "hello world";

// запись в файл
using (FileStream fstream = new FileStream(path,
    FileMode.OpenOrCreate))
{
    // преобразуем строку в байты
    byte[] input = Encoding.Default.GetBytes(text);
    // запись массива байтов в файл
    fstream.Write(input, 0, input.Length);
    Console.WriteLine("Текст записан в файл");
}
using (FileStream fstream = new FileStream(path,
    FileMode.OpenOrCreate))
{
    // заменим в файле слово world на слово house
    string replaceText = "house";
    fstream.Seek(-5, SeekOrigin.End); // минус 5 символов с конца
    потока
    byte[] input = Encoding.Default.GetBytes(replaceText);
    await fstream.WriteAsync(input, 0, input.Length);

    // считываем весь файл
    // возвращаем указатель в начало файла
    fstream.Seek(0, SeekOrigin.Begin);
    byte[] output = new byte[fstream.Length];
    await fstream.ReadAsync(output, 0, output.Length);
    // декодируем байты в строку
    string textFromFile = Encoding.Default.GetString(output);
    Console.WriteLine($"Текст из файла: {textFromFile}"); //
    hello house
}
```

Здесь также вначале записываем в файл строку "hello world". Затем также открываем файл и опять же перемещаемся в конец файла, не доходя до конца пять символов (то есть опять же с позиции символа "w"), и осуществляем запись строки "house". Таким образом, строка "house" заменяет строку "world".

Чтобы после этого считать весь файл, сдвигаем курсор на самое начало

```
fstream.Seek(0, SeekOrigin.Begin);
```

Консольный вывод программы:

```
Текст записан в файл
Текст из файла: hello house
```

Чтение и запись текстовых файлов. StreamReader и StreamWriter

Для работы непосредственно с текстовыми файлами в пространстве `System.IO` определены специальные классы: `StreamReader` и `StreamWriter`.

Запись в файл и StreamWriter

Для записи в текстовый файл используется класс `StreamWriter`. Некоторые из его конструкторов, которые могут применяться для создания объекта `StreamWriter`:

- `StreamWriter(string path)`: через параметр `path` передается путь к файлу, который будет связан с потоком
- `StreamWriter(string path, bool append)`: параметр `append` указывает, надо ли добавлять в конец файла данные или же перезаписывать файл. Если равно `true`, то новые данные добавляются в конец файла. Если равно `false`, то файл перезаписывается заново
- `StreamWriter(string path, bool append, System.Text.Encoding encoding)`: параметр `encoding` указывает на кодировку, которая будет применяться при записи

Свою функциональность `StreamWriter` реализует через следующие методы:

- `int Close()`: закрывает записываемый файл и освобождает все ресурсы
- `void Flush()`: записывает в файл оставшиеся в буфере данные и очищает буфер.
- `Task FlushAsync()`: асинхронная версия метода `Flush`
- `void Write(string value)`: записывает в файл данные простейших типов, как `int`, `double`, `char`, `string` и т.д. Соответственно имеет ряд перегруженных версий для записи данных элементарных типов, например, `Write(char value)`, `Write(int value)`, `Write(double value)` и т.д.
- `Task WriteAsync(string value)`: асинхронная версия метода `Write`. Обратите внимание, что асинхронные версии есть не для всех перегрузок метода `Write`.
- `void WriteLine(string value)`: также записывает данные, только после записи добавляет в файл символ окончания строки
- `Task WriteLineAsync(string value)`: асинхронная версия метода `WriteLine`

Рассмотрим запись в файл на примере:

```
string path = "notel.txt";
string text = "Hello World\nHello METANIT.COM";

// полная перезапись файла
```

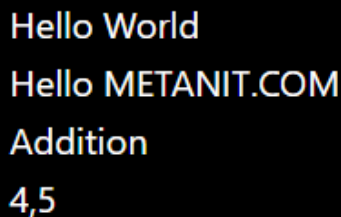
```

using (StreamWriter writer = new StreamWriter(path, false))
{
    await writer.WriteLineAsync(text);
}
// добавление в файл
using (StreamWriter writer = new StreamWriter(path, true))
{
    await writer.WriteLineAsync("Addition");
    await writer.WriteAsync("4,5");
}

```

В данном случае два раза создаем объект `StreamWriter`. В первом случае если файл существует, то он будет перезаписан. Если не существует, он будет создан. И в нее будет записан текст из переменной `text`. Во втором случае файл открывается для дозаписи, и будут записаны атомарные данные - строка и число.

По завершении в папке программы мы сможем найти файл `note.txt`, который будет иметь следующие строки:



```

Hello World
Hello METANIT.COM
Addition
4,5

```

В пример выше будет использоваться кодировка по умолчанию. но также можно задать ее явным образом:

```

using (StreamWriter writer = new StreamWriter(path, true,
System.Text.Encoding.Default))
{
    // операции с writer
}

```

Чтение из файла и `StreamReader`

Класс `StreamReader` позволяет нам легко считывать весь текст или отдельные строки из текстового файла.

Некоторые из конструкторов класса `StreamReader`:

- `StreamReader(string path)`: через параметр `path` передается путь к считываемому файлу
- `StreamReader(string path, System.Text.Encoding encoding)`: параметр `encoding` задает кодировку для чтения файла

Среди методов `StreamReader` можно выделить следующие:

- `void Close()`: закрывает считываемый файл и освобождает все ресурсы
- `int Peek()`: возвращает следующий доступный символ, если символов больше нет, то возвращает -1

- `int Read()`: считывает и возвращает следующий символ в численном представлении. Имеет перегруженную версию: `Read(char[] array, int index, int count)`, где `array` - массив, куда считываются символы, `index` - индекс в массиве `array`, начиная с которого записываются считываемые символы, и `count` - максимальное количество считываемых символов

- `Task<int> ReadAsync()`: асинхронная версия метода `Read`
- `string ReadLine()`: считывает одну строку в файле
- `string ReadLineAsync()`: асинхронная версия метода `ReadLine`
- `string ReadToEnd()`: считывает весь текст из файла
- `string ReadToEndAsync()`: асинхронная версия метода `ReadToEnd`

Сначала считаем текст полностью из ранее записанного файла:

```
string path = "note1.txt";
// асинхронное чтение
using (StreamReader reader = new StreamReader(path))
{
    string text = await reader.ReadToEndAsync();
    Console.WriteLine(text);
}
```

Считаем текст из файла построчно:

```
string path = "/Users/eugene/Documents/app/note1.txt";

// асинхронное чтение
using (StreamReader reader = new StreamReader(path))
{
    string? line;
    while ((line = await reader.ReadLineAsync()) != null)
    {
        Console.WriteLine(line);
    }
}
```

В данном случае считываем построчно через цикл `while`: `while ((line = await reader.ReadLineAsync()) != null)` - сначала присваиваем переменной `line` результат функции `reader.ReadLineAsync()`, а затем проверяем, не равна ли она `null`. Когда объект `sr` дойдет до конца файла и больше строк не останется, то метод `reader.ReadLineAsync()` будет возвращать `null`.

Бинарные файлы. BinaryWriter и BinaryReader

Для работы с бинарными файлами предназначена пара классов `BinaryWriter` и `BinaryReader`. Эти классы позволяют читать и записывать данные в двоичном формате.

BinaryWriter

Для создания объекта `BinaryWriter` можно применять ряд конструкторов. Возьмем наиболее простую:

```
BinaryWriter(Stream stream)
```

в его конструктор передается объект `Stream` (обычно это объект `FileStream`).

Основные методы класса `BinaryWriter`:

- `Close()`: закрывает поток и освобождает ресурсы
- `Flush()`: очищает буфер, дописывая из него оставшиеся данные в файл
- `Seek()`: устанавливает позицию в потоке
- `Write()`: записывает данные в поток. В качестве параметра этот метод

может принимать значения примитивных данных:

- `Write(bool)`
- `Write(byte)`
- `Write(char)`
- `Write(decimal)`
- `Write(double)`
- `Write(Half)`
- `Write(short)`
- `Write(int)`
- `Write(long)`
- `Write(sbyte)`
- `Write(float)`
- `Write(string)`
- `Write(ushort)`
- `Write(uint)`
- `Write(ulong)`

Либо можно передать массивы типов `byte` и `char`

- `Write(byte[])`
- `Write(char[])`
- `Write(ReadOnlySpan<byte>)`
- `Write(ReadOnlySpan<char>)`

При записи массива дополнительно можно указать, с кого элемента массива надо выполнять запись, а также число записываемых элементов массива:

- `Write(byte[], int, int)`
- `Write(char[], int, int)`

Рассмотрим простейшую запись бинарного файла:

```
string path = "person.dat";

// создаем объект BinaryWriter
using (BinaryWriter writer = new BinaryWriter(File.Open(path,
    FileMode.OpenOrCreate)))
{
    // записываем в файл строку
    writer.Write("Tom");
    // записываем в файл число int
    writer.Write(37);
    Console.WriteLine("File has been written");
}
```

Здесь в файл person.dat записываются два значения: строка «Tom» и число 37. Для создание объекта применяется вызов `new BinaryWriter(File.Open(path, FileMode.OpenOrCreate))`

Подобным образом можно сохранять более сложные данные. Например, сохраним в файл массив объектов:

```
string path = "people.dat";
// массив для записи
Person[] people =
{
    new Person("Tom", 37),
    new Person("Bob", 41)
};
using (BinaryWriter writer = new BinaryWriter(File.Open(path,
    FileMode.OpenOrCreate)))
{
    // записываем в файл значение каждого свойства объекта
    foreach (Person person in people)
    {
        writer.Write(person.Name);
        writer.Write(person.Age);
    }
    Console.WriteLine("File has been written");
}

class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

В данном случае последовательно сохраняем в файл `people.dat` данные объектов `Person` из массива `people`.

BinaryReader

Для создания объекта `BinaryReader` можно применять ряд конструкторов. Возьмем наиболее простую версию:

```
Reader(Stream stream)
```

в его конструктор также передается объект `Stream` (также обычно это объект `FileStream`).

Основные методы класса `BinaryReader`:

- `Close()`: закрывает поток и освобождает ресурсы
- `ReadBoolean()`: считывает значение `bool` и перемещает указатель на один байт
- `ReadByte()`: считывает один байт и перемещает указатель на один байт
- `ReadChar()`: считывает значение `char`, то есть один символ, и перемещает указатель на столько байтов, сколько занимает символ в текущей кодировке
- `ReadDecimal()`: считывает значение `decimal` и перемещает указатель на 16 байт
- `ReadDouble()`: считывает значение `double` и перемещает указатель на 8 байт
- `ReadInt16()`: считывает значение `short` и перемещает указатель на 2 байта
- `ReadInt32()`: считывает значение `int` и перемещает указатель на 4 байта
- `ReadInt64()`: считывает значение `long` и перемещает указатель на 8 байт
- `ReadSingle()`: считывает значение `float` и перемещает указатель на 4 байта
- `ReadString()`: считывает значение `string`. Каждая строка предваряется значением длины строки, которое представляет 7-битное целое число

С чтением бинарных данных все просто: соответствующий метод считывает данные определенного типа и перемещает указатель на размер этого типа в байтах, например, значение типа `int` занимает 4 байта, поэтому `BinaryReader` считывает 4 байта и переместит указатель на эти 4 байта.

Например, выше в примере с `BinaryWriter` в файл `person.dat` записывалась строка и число. Считаем их с помощью `BinaryReader`:

```
using (BinaryReader reader = new
BinaryReader(File.Open("person.dat", FileMode.Open)))
{
```

```

        // считываем из файла строку
        string name = reader.ReadString();
        // считываем из файла число
        int age = reader.ReadInt32();
        Console.WriteLine($"Name: {name}   Age: {age}");
    }

```

Конструктор класса `BinaryReader` также в качестве параметра принимает объект потока, только в данном случае устанавливаем в качестве режима `FileMode.Open`: `new BinaryReader(File.Open("person.dat", FileMode.Open))`.

В каком порядке данные были записаны в файл, в таком порядке мы их можем оттуда считать. То есть если сначала записывалась строка, а потом число, то в данном порядке мы их можем считать из файла.

Или подобным образом считаем данные из файла `people.dat`, который был записан в примере выше и который содержит данные объектов `Person`:

```

// список для считываемых данных
List<Person> people = new List<Person>();

// создаем объект BinaryWriter
using (BinaryReader reader = new
BinaryReader(File.Open("people.dat", FileMode.Open)))
{
    // пока не достигнут конец файла
    // считываем каждое значение из файла
    while (reader.PeekChar() > -1)
    {
        string name = reader.ReadString();
        int age = reader.ReadInt32();
        // по считанным данным создаем объект Person и добавляем
        в список
        people.Add(new Person(name, age));
    }
}
// выводим содержимое списка people на консоль
foreach(Person person in people)
{

```

Здесь в цикле `while` считываем данные. Чтобы узнать окончание потока, вызываем метод `PeekChar()`. Этот метод считывает следующий символ и возвращает его числовое представление. Если символ отсутствует, то метод возвращает `-1`, что будет означать, что мы достигли конца файла.

В цикле последовательно считываем значения для свойств объектов `Person` в том же порядке, в каком они записывались.

Работа с файлами в C++

Для работы с файлами в стандартной библиотеке определен заголовочный файл `fstream`, который определяет базовые типы для чтения и записи файлов. В частности, это:

- `ifstream`: для чтения с файла
- `ofstream`: для записи в файл
- `fstream`: совмещает запись и чтение

Для работы с данными типа `wchar_t` для этих потоков определены двойники:

- `wifstream`
- `wofstream`
- `wfstream`

Открытие файла

При операциях с файлом вначале необходимо открыть файл с помощью функции `open()`. Данная функция имеет две версии:

- `open(путь)`
- `open(путь, режим)`

Для открытия файла в функцию необходимо передать путь к файлу в виде строки. И также можно указать режим открытия. Список доступных режимов открытия файла:

- `ios::in`: файл открывается для ввода (чтения). Может быть установлен только для объекта `ifstream` или `fstream`
- `ios::out`: файл открывается для вывода (записи). При этом старые данные удаляются. Может быть установлен только для объекта `ofstream` или `fstream`
- `ios::app`: файл открывается для дозаписи. Старые данные не удаляются.
- `ios::ate`: после открытия файла перемещает указатель в конец файла
- `ios::trunc`: файл усекается при открытии. Может быть установлен, если также установлен режим `out`
- `ios::binary`: файл открывается в бинарном режиме

Если при открытии режим не указан, то по умолчанию для объектов `ofstream` применяется режим `ios::out`, а для объектов `ifstream` - режим `ios::in`. Для объектов `fstream` совмещаются режимы `ios::out` и `ios::in`.

```
std::ofstream out;           // поток для записи
out.open("D:\\hello1.txt"); // открываем файл для записи
```

```

std::ofstream out2;
out2.open("D:\\hello2.txt", std::ios::app); // открываем файл для
дозаписи

std::ofstream out3;
out2.open("D:\\hello3.txt", std::ios::out | std::ios::trunc); //
установка нескольких режимов

std::ifstream in;          // поток для чтения
in.open("D:\\hello4.txt"); // открываем файл для чтения

std::fstream fs;           // поток для чтения-записи
fs.open("D:\\hello5.txt"); // открываем файл для чтения-записи

```

Однако в принципе необязательно использовать функцию `open` для открытия файла. В качестве альтернативы можно также использовать конструктор объектов-потоков и передавать в них путь к файлу и режим открытия:

```

fstream(путь)
fstream(путь, режим)

```

При вызове конструктора, в который передан путь к файлу, данный файл будет автоматически открываться:

```

std::ofstream out("D:\\hello.txt");
std::ifstream in("D:\\hello.txt");
std::fstream fs("D:\\hello.txt", std::ios::app);

```

Вообще использование конструкторов для открытия потока является более предпочтительным, так как определение переменной, представляющей файловой поток, уже предполагает, что этот поток будет открыт для чтения или записи. А использование конструктора избавит от ситуации, когда мы забудем открыть поток, но при этом начнем его использовать.

В процессе работы мы можем проверить, открыт ли файл с помощью функции `is_open()`. Если файл открыт, то она возвращает `true`:

```

std::ifstream in;          // поток для чтения
in.open("D:\\hello.txt"); // открываем файл для чтения
// если файл открыт
if (in.is_open())
{
}

```

Заккрытие файла

После завершения работы с файлом его следует закрыть с помощью функции `close()`. Также стоит отметить, то при выходе объекта потока из области видимости, он удаляется, и у него автоматически вызывается функция `close`.

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream out;           // поток для записи
    out.open("D:\\hello.txt"); // открываем файл для записи
    out.close();                 // закрываем файл

    std::ifstream in;           // поток для чтения
    in.open("D:\\hello.txt");   // открываем файл для чтения
    in.close();                 // закрываем файл

    std::fstream fs;            // поток для чтения-записи
    fs.open("D:\\hello.txt");   // открываем файл для чтения-записи
    fs.close();                 // закрываем файл

    return 0;
}
```

Стоит отметить, что при компиляции через `g++` следует использовать флаг `-static`, если программа работает со файлами и использует типы из заголовочного файла `fstream`:

Чтение и запись текстовых файлов

Потоки для работы с текстовыми файлами представляют объекты, для которых не задан режим открытия `ios::binary`.

Запись в файл

Для записи в файл к объекту `ofstream` или `fstream` применяется оператор `<<` (как и при выводе на консоль):

```
#include <iostream>
#include <fstream>

int main()
{
```



```

std::ofstream out;           // поток для записи
out.open("D:\\hello.txt"); // открываем файл для записи
if (out.is_open())
{
    out << "Hello World!" << std::endl;
}

std::cout << "End of program" << std::endl;
return 0;
}

```

Данный способ перезаписывает файл заново. Если надо дозаписать текст в конец файла, то для открытия файла нужно использовать режим `ios::app`:

```

std::ofstream out("D:\\hello.txt", std::ios::app);
if (out.is_open())
{
    out << "Welcome to CPP" << std::endl;
}
out.close();

```

Чтение из файла

Если надо считать всю строку целиком или даже все строки из файла, то лучше использовать встроенную функцию `getline()`, которая принимает поток для чтения и переменную, в которую надо считать текст:

```

#include <iostream>
#include <fstream>
#include <string>
int main()
{
    std::string line;

    std::ifstream in("D:\\hello.txt"); // открываем файл для чтения
    if (in.is_open())
    {
        while (getline(in, line))
        {
            std::cout << line << std::endl;
        }
    }
    in.close(); // закрываем файл

    std::cout << "End of program" << std::endl;
    return 0;
}

```

Также для чтения данных из файла для объектов `ifstream` и `fstream` может применяться оператор `>>` (также как и при чтении с консоли):

```
#include <iostream>
#include <fstream>
#include <vector>

struct Operation
{
    int sum;          // купленная сумма
    double rate;      // по какому курсу
    Operation(double s, double r) : sum(s), rate(r)
    {}
};

int main()
{
    std::vector<Operation> operations = {
        Operation(120, 57.7),
        Operation(1030, 57.4),
        Operation(980, 58.5),
        Operation(560, 57.2)
    };

    std::ofstream out("D:\\operations.txt");

    if (out.is_open())
    {
        for (int i = 0; i < operations.size(); i++)
        {
            out << operations[i].sum << " " << operations[i].rate
<< std::endl;
        }
    }
    out.close();

    std::vector<Operation> new_operations;
    double rate;
    int sum;
    std::ifstream in("D:\\operations.txt"); // открываем файл для
чтения
    if (in.is_open())
    {
        while (in >> sum >> rate)
        {
            new_operations.push_back(Operation(sum, rate));
        }
    }
    in.close();
}
```

```

        for (int i = 0; i < new_operations.size(); i++)
        {
            std::cout << new_operations[i].sum << " - " <<
new_operations[i].rate << std::endl;
        }
        return 0;
    }

```

Здесь вектор структур Operation записывается в файл.

```

    for (int i = 0; i < operations.size(); i++)
    {
        out << operations[i].sum << " " << operations[i].rate <<
std::endl;
    }

```

При записи в данном случае будет создаваться файл в формате

```

120 57.7
1030 57.4
980 58.5
560 57.2

```

Используя оператор >>, можно считать последовательно данные в переменные sum и rate и ими инициализировать структуру.

```

while (in >> sum >> rate)
{
    new_operations.push_back(Operation(sum, rate));
}

```

Содержание отчета

Отчет должен включать:

- а) титульный лист;
- б) формулировку цели работы;
- в) описание результатов выполнения заданий:
 - листинги программ;
 - результаты выполнения программ;
- г) выводы, согласованные с целью работы.

Варианты

Вариант	Задание 1	Задание 2
1	1	20
2	2	19
3	3	18
4	4	17
5	5	16
6	6	15
7	7	14
8	8	13
9	9	12
10	10	11

Задания

1. Дан текстовый файл, содержащий целые числа. Удалить из него все четные числа.
2. В данном текстовом файле удалить все слова, которые содержат хотя бы одну цифру.
3. Дан текстовый файл. Создать новый файл, каждая строка которого получается из соответствующей строки исходного файла перестановкой слов в обратном порядке.
4. Дан текстовый файл. Создать новый файл, состоящий из тех строк исходного файла, из чисел которых можно составить арифметическую прогрессию.
5. Даны два текстовых файла, содержащие целые числа. Создать файл из различных чисел, которые содержатся: а) в каждом исходном файле; б) только в одном из двух исходных файлов; в) только в первом исходном файле; г) хотя бы в одном из двух исходных файлов.
6. Создать и заполнить файл случайными целыми значениями. Выполнить сортировку содержимого файла по возрастанию.

7. Создать типизированный файл записей со сведениями о телефонах абонентов; каждая запись имеет поля: фамилия абонента, год установки телефона, номер телефона. По заданной фамилии абонента выдать номера его телефонов. Определить количество установленных телефонов с N-го года.

8. В текстовый файл занесены пары чисел, разделенных пробелом (каждая пара чисел – в новой строке). Рассматривая каждую пару как координаты точек на плоскости, найти наибольшее и наименьшее расстояния между этими точками.

9. Имеется файл с текстом. Осуществить шифрование данного текста в новый файл путем записи текста в матрицу символов по строкам, а затем чтение символов из этой матрицы по столбцам. Осуществить расшифровку полученного текста.

10. Создать программу, переписывающую в текстовый файл g содержимое файла f, исключая пустые строки, а остальные дополнить справа пробелами или ограничить до n символов.

11. В файле, содержащем фамилии студентов и их оценки, изменить на прописные буквы фамилии тех студентов, которые имеют средний балл за национальной шкалой более «4».

12. Из текстового файла удалить все слова, содержащие от трех до пяти символов, но при этом из каждой строки должно быть удалено только четное количество таких слов.

13. Получить файл g, состоящий из строк файла f, содержащих заданную строку S. Предусмотреть случай, когда строка размещается в двух строках файла «с переносом».

14. Получить файл g, в котором текст выровнен по правому краю путем равномерного добавления пробелов.

15. Из текста программы выбрать все числа (целые и вещественные) и записать их в файл g в виде: число 1 – номер строки, число 2 – номер строки и так далее.

16. Определить, симметричен ли заданный во входном файле текст.

17. Текстовый файл содержит записи о телефонах и их владельцах. Переписать в другой файл телефоны тех владельцев, фамилии которых начинаются с букв К и С.

18. В файле содержится совокупность текстовых строк. Изменить первую букву каждого слова на заглавную.

19. В файле содержится текстовая строка. Определить частоту повторяемости каждой буквы в тексте и вывести ее.

20. Дан текстовый файл со статистикой посещения сайта за неделю. Каждая строка содержит ip адрес, время и название дня недели (например, 139.18.150.126 23:12:44 sunday). Создайте новый текстовый файл, который бы содержал список ip без повторений из первого файла. Для каждого ip укажите количество посещений в неделю, наиболее популярный день недели, наиболее популярный отрезок времени длиной в один час. Последней строкой в файле добавьте наиболее популярный отрезок времени в сутках длиной один час в целом для сайта.