# Elastic File

## Objective

To implement in C++ for Windows (or C# depending on your skills) a class overriding the standard file operations (Open/Create, Read, Write, Move Cursor, Truncate and Close) in such a way that when data is written in the middle of the file it gets inserted without overwriting the existing file data following the cursor position. Similarly the file can be truncated not only from its end, but also at any random position within a file.

All the implemented operations must encapsulate the underlying logic and provide the user with a regular file access experience.

## Caveats

The files created and accessed with this library can be quite big, this is why the file must not be overwritten every time when the data is inserted or deleted. In other words, the amount of excessive (supplementary) information being written must depend only on the size of the serialized data, but NOT on the current size of the file itself.

The implementation must not use any third party libraries or solutions, but rather rely on the standard system file access routines (like fopen, fread, fwrite etc).

## Class Methods

| Method | Description |
|---|---|
| `FileHandle FileOpen(string FileName, OpenMode Mode);` | Opens (or Creates) a file with a given *FileName* using the preferred access *Mode*.<br><br>The access *Mode* can be a combination of flags which must semantically repeat those from the .NET FileMode enumeration.<br><br>Returns a file handle on success or Invalid file handle in case of error. |
| `bool FileSetCursor(FileHandle File, ULong Offset, CursorMoveMode Mode);` | Move the file cursor to the given *Offset*. The *Offset* can be counted from the beginning of the file, current cursor position or from the end of the file. Check the SeekOrigin enumeration and declare your own enumeration with similar values (will be passed as *Mode* parameter).<br><br>* If the new cursor position is beyond the data which this file contains, the file must be extended up to the required size.<br><br>Returns True on success, False otherwise. |
| `ULong FileGetCursor(FileHandle File);` | Returns the current file cursor offset from the beginning of the file.<br><br>When the cursor is positioned at the end of the file this method must return the size of the data which this file contains. |
| `ULong FileRead(FileHandle File, ByteArray Buffer, ULong Size);` | Read a block of bytes of the requested *Size* from the *File*, starting from the current cursor position, and put the result into a user allocated memory block passed as *Buffer* (PBYTE in case of C++).<br><br>* If the actual amount of data read is less than the requested Size it must be handled gracefully.<br><br>Returns the number of bytes actually read from the file into the *Buffer*. |
| `ULong FileWrite(FileHandle File, ByteArray Buffer, ULong Size, bool Overwrite);` | Write a block of bytes of the requested *Size* to the *File* at the current cursor position. The data to be written is given as a reference to a memory block passed as *Buffer* (PBYTE in case of C++).<br><br>If the *Overwrite* parameter is True then the data is written in a standard way (overwriting bytes following the current cursor position).<br><br>If *Overwrite* parameter is False then the file must get expanded (stretched) by the required number of bytes at the cursor position, thus the data is actually inserted into the file rather then written over.<br><br>Returns the number of bytes from the *Buffer* actually written to the file. |

| | |
|---|---|
| `bool FileTruncate(FileHandle File, ULong CutSize);` | Removes the number of bytes passed as the *CutSize* parameter, starting from the current cursor position.<br><br>If the cursor is positioned somewhere inside the file, then the data must still be cut from within the file so that the subsequent reads will not get it.<br><br>Returns True on success, False otherwise. |
| `bool FileClose(FileHandle File);` | Closes the file using its handle passed with *File* parameter.<br><br>Returns True on success, False otherwise. |

## Tests

The following test scenario must be implemented using the above functionality:

1. User inputs three numbers: the length for the first pass (**m**), the length for the second pass (**M**) and the number of iterations (**N**)
2. At the first pass your application must create a new file with N records, each having a length of **m**, written consequently, as follows [m][m][m]...[m]. The records must be random alphabetical strings (a-zA-Z).
3. At the second pass the application must insert another N records of length of M bytes each in between the records from the first pass, as follows: [m][M][m][M][m][M]...[m][M]. The records must again be random alphabetical strings (a-zA-Z).
4. The next step is to read every 10th record from the file and output the maximum string (alphabetical order)
5. The final test is to delete every even pair [m][M] from the file, i.e ~~[m][M]~~[m][M]~~[m][M]~~[m][M]...

After each test the application must output the time in milliseconds which this test has taken (calculated using GetTickCount or any other system counter), the current size of the physical file and amount of data which it stores.

## Deployment

The result of you work must be delivered as a regular MS Visual Studio solution containing the File Library (class) and the test console application implementing the Tests listed above.

## Criteria

We will generate a file of ~4GB using your library and consider your solution based on:

1. Time needed to perform the random reads of data which totals ~100MB.
2. Time needed to randomly write the data totaling ~100MB.
3. Time needed to perform 10 random truncate operations with the resulting file to become twice as small.
4. The overall efficiency of your storage.

## Questions

If anything stays undefined or unclear from this assignment, then instead of asking, you shall just implement it in any preferred way and provide a note of what was unclear and what were the assumptions which you had for that.