**WinFile**

Anything 😊 you can do with a MS-Windows file.

This little project started when I decided to put all functions that perform some action on a file together in one object oriented class. Why? Well, in larger projects all sorts of functions on files pop up along the way. Some can deal with a “FILE\*” pointer and some have to deal with a Windows kernel HANDLE. You can use the old style POSIX functions like “fopen” (more likely: “fopen\_s”) and “fgets”, “fputs” or “getc” with a standard FILE pointer. But some other functions like attributes or file times require a kernel HANDLE that can be obtained through the “CreateFile” SDK function, instead of using the “fopen” counterparts.

At other times you might want to use the standard library ifstream or ofstream interfaces, and that leaves you with yet a third type of interface to a file. At times it gets quite confusing. And while there are so many subfunctions, the handling routines to these functions tend to get quite scattered through the code.

**WinFile as a function collection.**

The “WinFile” project started as a collection of functions of things you can do with a file on the MS-Windows system. So that it stays all together. Through time it grew and collected more functions and tricks. OK, You can simply call a SDK function like “CopyFile” to copy a file, but do you (or I) recollect all extra options you can pull out of your sleeve if you use “CopyFileEx”? The WinFile class collects all of those tricks. Do I need a FILE or a HANDLE for this function? Where have I seen this trick before?

Here it’s all together in one interface.

**The best documentation**

is of course the interface file in the project “WinFile.h”. You can find all the methods and enum-ed parameters to the methods there. So go take a look at that file!

Here is a sort of concise recapitulation on what you can do with a WinFile:

* Opening in test/binary mode with or without sharing or buffering
* Reading and writing in text or binary mode
* Reading and writing single characters and un-getting a character
* File positioning for random access
* Moving, copying, deleting, deleting to the trashcan
* Testing for existence, read access and / or write access
* Creating temporary files
* Getting and setting attributes
* Getting and setting file times
* Locking and unlocking the file or portions of the file
* Manipulations of the file name and getting file names through a dialog
* Directory operations like creating a path and ensuring the existence
* Using as a shared memory virtually mapped file
* Using the CSIDL\_\* special folder names of the Windows shell
* Error handling and getting the last error as a string
* Defusing a BOM (byte-order-mark) at the beginning of a file

**Performance and caching**

Along the way I hit a snag. Performance! What a bummer to find that reading a string in text translation mode – where you translate “\r\n” in a file to a single “\n” – tended to be quite slow. More than 10 times slower in my first version that say “fgets(buffer,count,FILE)”. That wasn’t going to happen I said to myself! So why was it so slow?

Well, of course when you open a file with “fopen” or “fopen\_s” and get a FILE pointer back, some caching is done in the background. It’s always infinitely faster to cache a part of a file, than to read every single character from a file HANDLE.

So I rolled up my sleeves and started to program my own buffer cache in the WinFile class. The first version of that cache could easily read in a buffer for reading and return cached versions of strings. Wow. That was a lot faster, even on a SSD system. And even faster on a real HDD with moving arms above rotating platters.

The real challenge proved to be, not to be able to also write to the cache, but to synchronize the case in case we are doing random access to the file. Reading and writing in random access mode means that the ‘real’ file pointer of the operating system is not in sync with the file pointer in the cached buffer. So we must adjust at all times to flush the buffer and re-sync the file pointer.

**Speed comparison**

After everything worked functionally correct, the curiosity on the comparison with other file access methods piqued my mind. How did my newborn baby compared to say the ‘ifstream’ interface?

Well: here is a comparison table (compiled from the test program in “main.cpp”

|  |  |  |
| --- | --- | --- |
| Test | SSD | HDD |
| WinFile Reading 3MB chars | 0.0346218 | 0.239968 |
| FILE\* Reading 3MB chars | 0.0534943 | 0.517571 |
| WinFile Reading 3MB strings | 0.0061391 | 0.063808 |
| FILE\* Reading 3MB strings | 0.0142861 | 0.143153 |
| Ifstream Reading 3MB strings | 0.0204958 | 0.190012 |

(Times measured with HPFCounter from <https://github.com/edwig/marlin>)

Apart from the fact that SSD’s are faster than HDD’s (sic) what we can learn from this test is that the created WinFile cache is considerably faster than reading from “fgets” FILE’s (at least two times faster) and that it is also a factor 3 faster than the standard library ifstream (!). And those times are quite consistent when you run the test multiple times.

Is that a conclusive comparison? Most certainly not. Much can be done to twiddle with the results, the drives, the sort of file we are reading or writing and so on. Lovers of the standard library will point out that the buffer size of an ifstream can be increased in order to get better performance. You can do so yourself by using the (ifstream).rdbuf()->pubsetbuf(buf,size) method. And while this DOES give a better read performance under Unix like operating systems like Apple-OS and Linux, it does nothing on MS-Windows. But somebody might find a surprise just yet…

**It’s still growing**

The WinFile class is still growing. Every now and then I find a trick or a function in one of the projects I’m working on professionally and add it to this class.

The last version can be found at: <https://github.com/edwig/WinFile>