I implemented a CPP, and two C programs by referencing the recitation notes. In the implementation of mmapc.c, instead of

```
ssize_t n = write(1, ptr, size);
```

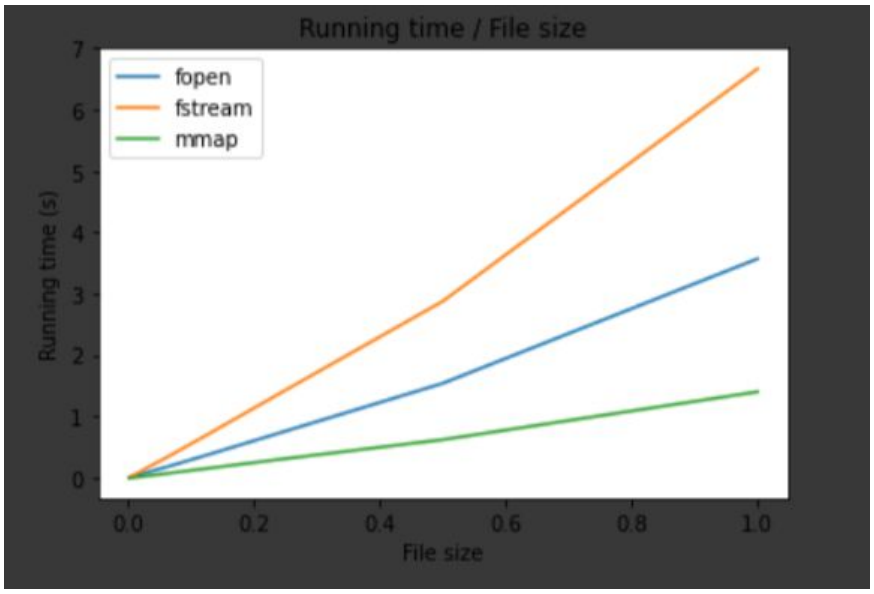
as we did in recitation, I use

```
ssize_t n = read(1, ptr, size);
```

because this speeds up the mmap function due to the fact that the write function causes so much wasted time for writing the content of the file to console. But there is only one drawback of using the read function that you may have to ***press enter one more time*** to see the result after you execute the executable file produced at the compilation. Please note that during the grading.

With the loremipsum.txt file provided, the fstream lasted 6.66 seconds to process the loremipsum.txt, fopen lasted 3.57 seconds and the mmap function that uses read() instead of write() lasted 1.40 seconds!

I made a slight performance optimization on the fstream implementation by using << operator instead of getChar() because << operator automatically gets rid of the spaces, tabs and etc. I could do that because the task was only counting the number of occurrences of character ‘a’.



Even with this optimization, the slowest one among all three implementations is the fstream and this is also true for smaller file sizes. Since I wanted to measure the impact of the size of the file, I created 9 more files that each file halves the size of the previous file. Let’s say we have x number of characters in loremipsum.txt, then in the second file I had x/2 characters, in the third one I had x/4 characters, etc. In a nutshell, I reduced the number of characters in the file by a factor of 2 just to observe the difference between the execution times. Then, for each implementation, I execute these 10 different text files of different sizes. But the running time complexities cannot be precisely understood from the graph because the elapsed time between the start and end of the execution is measured as very very fast when the size of the file is reduced by a certain factor. With these particular file sizes, it seems like all functions run in linear time but this observation may be change while the size of the files gets bigger. On the other hand, it was not really practical for me to double the file size at each time to measure the elapsed time because the original file provided was 266 MB itself, which is very big. But with this graph above, it is observable that there is a significant time difference stemming from the function’s nature. Also for the mmap function, the smallest 4 files got an execution time of approximately 0 while the fopen and fstream can get the 0 execution time only for their smallest one file. This is not very visible from the

graph, because there is a really small difference between the small numbers mathematically. That's why I'm also inserting a table for the functions and execution times in seconds for different file sizes.

	fstream	fopen	mmap
loremipsum.txt	6.67	3.57	1.40
½ (Half of the size of loremipsum.txt)	2.88	1.54	0.62
¼ (Quarter of the size of the loremipsum.txt)	1.43	0.76	0.31
1/8	0.7	0.37	0.15
1/16	0.34	0.18	0.07
1/32	0.17	0.09	0.03
1/64	0.08	0.04	0.01
1/128	0.04	0.02	0
1/256	0.02	0.01	0
1/512	0.01	0.01	0
1/1024	0	0	0

According to my research, indeed the slowest I/O file handler is C++ that proves the experimental results because the fstream in c++ is implemented as a wrapper for C FILE I/O. Because of the different classes their functions and subfunctions belong to, the indirectness of the execution slows down the fstream execution, while C FILE I/O directly operates on the files without any directions between classes and wrapper code.

Then I made a research about why the mmap function is faster than the classical C FILE I/O that which handles the files as streams. It has some pros and cons. Handling the small files with the mmap function does not really make sense because this explicit mapping creates some overhead which is unnecessary for reading a small file. But when the file in consideration is a large file, then this creates an advantage over the regular streams in C because the memory pages are the only pages that are going to be read. And also it can handle very large files because it implements the idea of virtual memory by letting the operating system to decide the allocation of bytes in physical memory. And lastly, not for this case but generally if the same file would be processed by different processes then mapping creates a significant performance optimization because this means all processes can access the content from the memory instead of reading the file to their private buffers. But there is a small disadvantage for the developer side that the implementation does not make much sense due to different types of new variables and error handling, and reading as a stream is much easier to implement but this is just my personal opinion.