

**Source Code:** Please see zip file.

**Correctness:** See below for execution output screenshots that verifies correctness of implementation and compares my implementation with the Unix-original stdio.h.

**Step 5:** Test your implementation of stdio.h using the eval executable by running the following commands.

```
erika@erika:~/Desktop/prog3$ ./eval r u a hamlet.txt
Reads : Unix I/O [Read once] = 132
erika@erika:~/Desktop/prog3$ ./eval r u b hamlet.txt
Reads : Unix I/O [Block transfers] = 168
erika@erika:~/Desktop/prog3$ ./eval r u c hamlet.txt
Reads : Unix I/O [Char transfers] = 145426
erika@erika:~/Desktop/prog3$ ./eval r u r hamlet.txt
bash: ./eval: No such file or directory
erika@erika:~/Desktop/prog3$ ./eval r u r hamlet.txt
Reads : Unix I/O [Random transfers] = 443
erika@erika:~/Desktop/prog3$ ./eval r f a hamlet.txt
Reads : C File I/O [Read once] = 8220
erika@erika:~/Desktop/prog3$ ./eval r f b hamlet.txt
Reads : C File I/O [Block transfers] = 6293
erika@erika:~/Desktop/prog3$ ./eval r f c hamlet.txt
Reads : C File I/O [Char transfers] = 2497
erika@erika:~/Desktop/prog3$ ./eval r f r hamlet.txt
Reads : C File I/O [Random transfers] = 5980
erika@erika:~/Desktop/prog3$ ./eval w u a test.txt
Writes: Unix I/O [Read once] = 368
erika@erika:~/Desktop/prog3$ ./eval w u b test.txt
Writes: Unix I/O [Block transfers] = 392
erika@erika:~/Desktop/prog3$ ./eval w u c test.txt
Writes: Unix I/O [Char transfers] = 227456
erika@erika:~/Desktop/prog3$ ./eval w u r test.txt
Writes: Unix I/O [Random transfers] = 16791
erika@erika:~/Desktop/prog3$ ./eval w f a test.txt
Writes: C File I/O [Read once] = 3739
erika@erika:~/Desktop/prog3$ ./eval w f b test.txt
Writes: C File I/O [Block transfers] = 2007
erika@erika:~/Desktop/prog3$ ./eval w f c test.txt
Writes: C File I/O [Char transfers] = 1123
erika@erika:~/Desktop/prog3$ ./eval w f r test.txt
Writes: C File I/O [Random transfers] = 3760
erika@erika:~/Desktop/prog3$
```

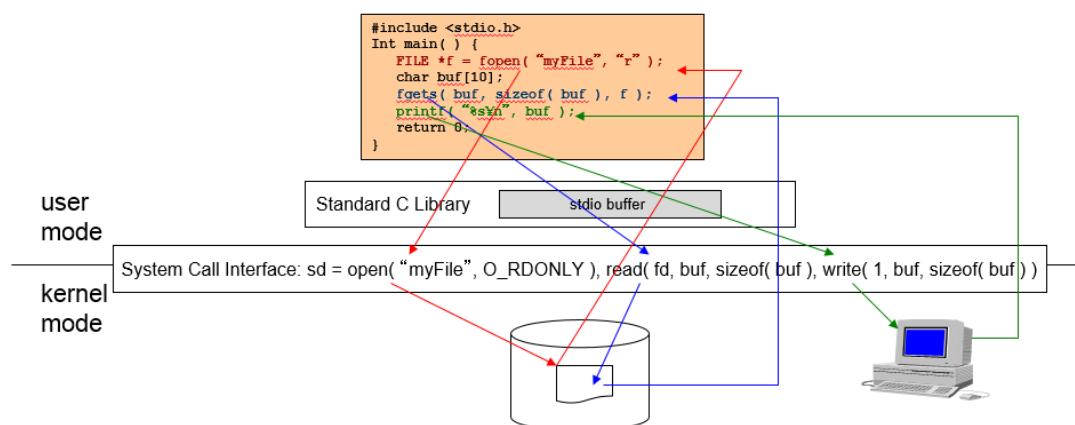
**Step 6:** Replace the first line of the eval.cpp, (i.e., "stdio.h") with <stdio.h> to use the Unix-original stdio.h rather than your own, recompile it with "./compile.sh", and rerun the eval program with the following test cases:

```
erika@erika:~/Desktop/prog3$ ./eval r f a hamlet.txt
Reads : C File I/O [Read once] = 265
erika@erika:~/Desktop/prog3$ ./eval r f b hamlet.txt
Reads : C File I/O [Block transfers] = 187
erika@erika:~/Desktop/prog3$ ./eval r f c hamlet.txt
Reads : C File I/O [Char transfers] = 2564
erika@erika:~/Desktop/prog3$ ./eval r f r hamlet.txt
Reads : C File I/O [Random transfers] = 337
erika@erika:~/Desktop/prog3$ ./eval w f a test.txt
Writes: C File I/O [Read once] = 294
erika@erika:~/Desktop/prog3$ ./eval w f b test.txt
Writes: C File I/O [Block transfers] = 383
erika@erika:~/Desktop/prog3$ ./eval w f c test.txt
Writes: C File I/O [Char transfers] = 1709
erika@erika:~/Desktop/prog3$ ./eval w f r test.txt
Writes: C File I/O [Random transfers] = 537
erika@erika:~/Desktop/prog3$
```

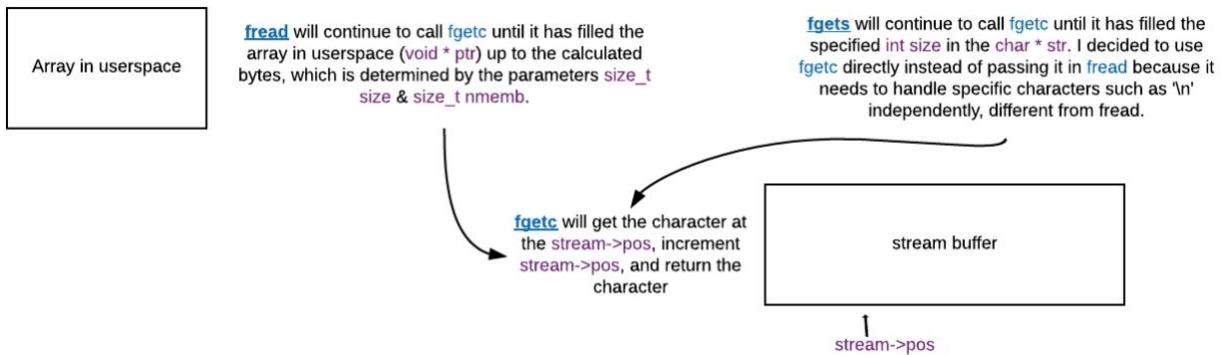
\*Screenshots are cropped for readability, full window screenshot is available in the zip file.

←----- START OF REPORT -----→

**Documentation:** of your stdio.cpp implementation including explanations and illustration in one or two pages.

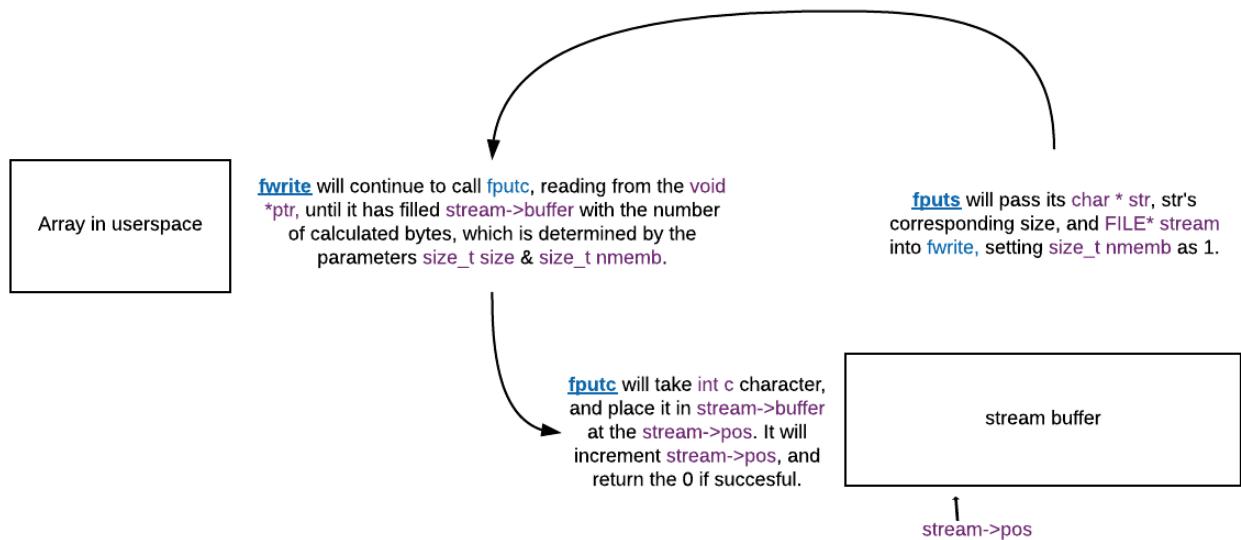


**Purpose:** The purpose of this programming assignment is to design and implement a core input and output class similar to the C/C++ standard I/O library, stdio.h. In this program, we will learn the importance of a buffer to help reduce the number of read and write system calls with file streams.



**Implementation of read methods:** For the read methods, I decided to call each other to reduce code redundancy, make the code easier to understand, and to help reduce the number of system calls made, in this case, the read system call.

**Implementation of buffer use in regard to read methods:** In this program, we are utilizing the stream buffer to create a caching system. This is the main method of avoiding excessive system calls. However, it is possible that the buffer is smaller than the total bytes needing to be read by `fread()`. In this case, we have to consider what to do when the `stream->pos` position reaches the end of the stream buffer but still has more bytes to read. The code takes advantage of the `fpurge()` method. Once the stream buffer is filled, it will write the data into the array in userspace, call `purge` to clear the buffer, and then refill the buffer so that `fgetc()` can restart at position 0 and continue to get characters.



**Implementation of write methods:** For the read methods, I decided to follow a similar approach to the read methods. Again, the purpose of calling each other is mostly to increase code clarity, reduce code redundancy, and help reduce number of system calls made. The write methods were actually slightly simpler to implement because I was able to pass `fputs()` through `fwrite()`.

*Implementation of buffer use in regard to write methods:* Unlike the read methods, specifically fgetc, fputc does not perform any system call directly. Instead, it calls flush(). Flush() is called when fputc() realizes that stream->pos is at the end of the buffer. This means that it will need to write the buffer contents into the device / output, and then clear it to allow for more characters to be written by fputc. However, similar to implementation in regard to read methods, this also helps reduce the number of system calls made.

*Implementation of stream's last operation variable:* The way to mitigate confusion between the data available after write and read is to use stream->lastop variable. In order to use this, we must change stream->lastop between reads and writes, and perform fpurge() and fflush() accordingly

#### Discussions in one or two pages.

- Performance consideration between your own stdio.h and Unix I/O

Command	Description	Unix I/O (u)	Own stdio.h (f)
.eval r _ a hamlet.txt	Read hamlet.txt at once	132	8220
.eval r _ b hamlet.txt	Read hamlet.txt every 4096 bytes	168	6293
.eval r _ c hamlet.txt	Read hamlet one by one character	145426	2497
.eval r _ r hamlet.txt	Read hamlet with random sizes	443	5980
.eval w _ a hamlet.txt	Write to test.txt at once	368	3739
.eval w _ b hamlet.txt	Write to test.txt every 4096 bytes	392	2007
.eval w _ c hamlet.txt	Write to test.txt one by one character	227456	1123
.eval w _ r hamlet.txt	Write to test.txt with random sizes	16791	3760

*Unix is faster:* The results in the above table makes sense. Being able to call system calls directly in Unix I/O, by not having to refer to or use a buffer should dramatically improve performance. That is what can be observed with the above numbers. My own stdio.h performance time was dramatically much larger than the Unix I/O calls. This is not only due to the presence of the buffer but also handling of the buffer's many variables.

*My one by one character is the most efficient:* What was surprising was that, opposite to the UNIX I/O, both my read and write functions were most optimal when reading character by character. However, this is likely due to the implementation of my program. Several methods build on top of each other. For example, fputs() calls fwrites() which calls fputc(). It only makes sense that the fputc() will be the fastest out of all of them because it goes through the least number of calls.

- Performance consideration between your own stdio.h and the Unix-original stdio.h

Command	Description	<stdio.cpp>	"stdio.cpp"
.eval r f a hamlet.txt	Read hamlet.txt at once	265	8220
.eval r f b hamlet.txt	Read hamlet.txt every 4096 bytes	187	6293
.eval r f c hamlet.txt	Read hamlet one by one character	2564	2497
.eval r f r hamlet.txt	Read hamlet with random sizes	337	5980
.eval w f a hamlet.txt	Write to test.txt at once	294	3739
.eval w f b hamlet.txt	Write to test.txt every 4096 bytes	383	2007
.eval w f c hamlet.txt	Write to test.txt one by one character	1709	1123

.eval w f r hamlet.txt	Write to test.txt with random sizes	537	3760
------------------------	-------------------------------------	-----	------

*Unix-original stdio.h is faster:* Unfortunately, my implementation of stdio.cpp is not nearly as efficient as Unix. While I'd confess that the lack of performance on my stdio.cpp is likely due to my inexperience and lack of expertise in algorithm efficiency. I do believe that the buffer size may also have an effect on performance. I am not sure what the size or type of buffer Unix is using, however, I imagine that the larger buffer or more efficient data structure are used, the better the performance.

*My one by one character is most efficient:* Again, my character by character handling is the most efficient, likely due to the same reason as explained in previous section. My structure of the methods and how they call each other is likely contributing to this lack of performance.

- Limitation and possible extension of your program

After reviewing my performance compared to the Unix I/O and their stdio.cpp implementation, I am curious to whether making the following changes will improve my performance:

- *The structure of buffer:* Currently we are using an array as a buffer structure. Since we are not performing random access on the array, I am curious to whether a queue would work as well. In fact, it may be better since we are the buffer in a FIFO manner. And on average, insertion and deletion of buffer is O(1) time complexity compared to array's O(n). We would also need less variables to keep track of position and size.
- *How the methods call each other:* In my implementation, methods call the "smallest" methods such as fgetc and fputc, as these methods are the ones that deal closely with the buffer purging and flushing. However, I wonder if instead, flush and purge were performed in fwrite and fread instead and have fputc and fgetc call them respectively. This may result in performance that is more aligned with Unix's stdio and I/O.