University of Washington Bothell

CSS503: System Programming

Program 3

C++ Standard I/O Library

1. Purpose

In this programming assignment you will design and implement your own core input and output functions of the C/C++ standard I/O library: stdio.h.

2. Linux I/O

Unix provide system calls for file I/O such as open(), read(), write(), and lseek(). However, these system calls are not supported on non-Unix based systems such as Window. Therefore, the C/C++ standard I/O library was created so that applications which require these functions can be easily ported through recompilation across these systems.

3. C/C++ Standard I/O Library Overview

The standard I/O library is an architecture independent library that allows C/C++ programs to read and write files instead of directly calling the underlying OS system calls.  The library's functions add buffering and provide a user-friendly file stream interface (FILE \*).  The file stream interface is implemented by the standard I/O library utilizing the underlying system calls.  When reading and/or writing small byte-counts (e.g., reading one line at a time from a file), buffered functions are faster.

The read() and write() system calls operate on file descriptors and read/write from/to buffers not strings.  A file descriptor is just an integer referring to a currently open file.  The OS uses that number as an index into the file descriptor table of files currently in use to access the actual device (e.g., disk, network, terminal).

On the other hand, file streams operators (fread/fwrite) interact directly with file streams: FILE \*.   File streams are dynamically allocated and allow reading/writing raw data. File stream operators, fread() and fwrite(), use the type void \* since there are no data-specific requirements.

The core input and output functions defined in <stdio.h> include:

|  |  |
| --- | --- |
| **Function name** | **Description** |
| fopen | opens a file |
| fflush | synchronizes an output stream with the actual file |
| setbuf, setvbuf | sets the size of an input/output stream buffer |
| fpurge | clears an input/output stream buffer |
| fread | reads from a file |
| fwrite | writes to a file |
| fgetc | reads a character from a file stream |
| fputc | writes a character to a file stream |
| fgets | reads a character string from a file stream |
| fputs | writes a character string to a file stream |
| fseek | moves the file position to a specific location in a file |
| feof | checks for the end-of-file |
| fclose | closes a file |
| printf | prints formatted output to stdout |

Figure 1 below shows how the standard I/O library utilizes a buffer to reduce the number of read and write system calls with filestreams.

|  |
| --- |
| C:\Users\dimpsey\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\154D375D.tmp |
| Figure 1. This figure shows Standard I/O functions using FILE instead of file descriptor. |

4. FILE Data Structure and fopen()

Upon a file open, fopen() returns a pointer to a FILE object that maintains the attributes of the opened file.

On canvas is posted a version of the header file we will use for the project. The following shows the class FILE definition:

#ifndef \_MY\_STDIO\_H\_

#define \_MY\_STDIO\_H\_

#define BUFSIZ 8192 // default buffer size

#define \_IONBF 0 // unbuffered

#define \_IOLBF 1 // line buffered. Do not need to implement this mode.

#define \_IOFBF 2 // fully buffered

#define EOF -1 // end of file

class FILE

{

public:

FILE() :

fd(0), pos(0), buffer((char \*)0), size(0 , actual\_size(0),

mode(\_IONBF), flag(0), bufown(false), lastop(0), eof(false) {}

int fd; // a Unix file descriptor of an opened file

int pos; // the current file position in the buffer

char \*buffer; // an input or output file stream buffer

int size; // the buffer size

int actual\_size; // actual buffer size when read() returns # bytes smaller than size

int mode; // \_IONBF, \_IOLBF, \_IOFBF. You do not need to implement \_IOLBF.

int flag; // O\_RDONLY

// O\_RDWR

// O\_WRONLY | O\_CREAT | O\_TRUNC

// O\_WRONLY | O\_CREAT | O\_APPEND

// O\_RDWR | O\_CREAT | O\_TRUNC

// O\_RDWR | O\_CREAT | O\_APPEND

bool bufown; // true if allocated by stdio.h or false by a user

char lastop; // 'r' or 'w'

bool eof; // true if EOF is reached

};

#include "stdio.cpp"

#endif

When opening a file, the fopen() function receives not only the file name to open but also various file access modes:

**r** Open text file for reading.

**r+** Open for reading and writing.

**w** Truncate file to zero length or create text file for writing.

**w+** Open for reading and writing. The file is created if it does not exist, otherwise truncated.

**a** Open for appending (writing at end of file). The file is created if it does not exist.

**a+** Open for reading and appending (writing at end of file). The file is created if it does not exist. The initial file position for reading is at the beginning of the file, but output is always appended to the end of the file.

The fopen() function must:

* instantiate a FILE object
* initialize it according to the file modes
* allocate a file stream buffer within the FILE object
* open a file using the corresponding OS system call, (e.g., open in Unix)

Binary vs. text files:  Linux doesn't differentiate between text and binary files, so 'b' type for fopen() has no effect

|  |
| --- |
| C:\Users\dimpsey\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\47505E73.tmp |
| Figure 2. Differences between file-opening modes using stdio.h functions vs. system calls. |

5. Our stdio.cpp File

In addition to stdio.h, you will also find stdio\_template.cpp on Canvas to support with this program. The file stdio\_template.cpp has already implemented: **printf, setvbuf, setbuf, fopen,**and **feof**.

Note that printf accepts only %d, and that the other functions are partially implemented -- just enough to be able to run the driver and performance test programs. No need to implement the rest of the function.

Also on canvas are the files driver.cpp and eval.cpp. These should not be modified and are only used to test your implementation of stdio. They should include “stdio.h”; they don't need to be aware of the existence of “stdio.cpp”.  Note that “stdio.cpp” is included at the bottom of “stdio.h”, so that it is possible to compile a user program like this:

% g++ driver.cpp

% g++ eval.cpp

6. Statement of Work

**Step 1:** Copy the source code and compile script from canvas to a Linux system.

**Files**: compile.sh, eval.cpp, driver.cpp, stdio.h and stdio\_template.cpp

**Step 2:** Rename stdio\_template.cpp to stdio.cpp. Implement all missing functions; noted by, “//complete it” in the file.

**Step 3:**Build the eval and driver executables using the compile.sh build script.

**Step 4:** Test your implementation of stdio.h using the driver executable and the texts provided on canvas: hamlet.txt, othello.txt

* Execute: ./driver hamlet.txt > output\_hamlet.txt

Compare output\_hamlet.txt, test1.txt, test2.txt, test3.txt to versions on canvas

* Execute: ./driver othello.txt > output\_othello.txt

Compare output\_othello.txt, test1.txt, test2.txt, test3.txt to versions on canvas

**Step 5:** Test your implementation of stdio.h using the **eval** executable to test different read/write combinations and sizes. The set of tests which should be run can be executed by running the eval\_tests.sh script.

* Execute: ./eval\_tests.sh > eval\_test\_out.txt

The tests which are run are the following:

./eval r u a hamlet.txt read hamlet.txt with unix I/O at once.

./eval r u b hamlet.txt read hamlet.txt with unix I/O every 4096 bytes.

./eval r u c hamlet.txt read hamlet.txt with unix I/O one by one character.

./eval r u r hamlet.txt read hamlet.txt with unix I/O with random sizes.

./eval r f a hamlet.txt read hamlet.txt with your stdio.cpp at once.

./eval r f b hamlet.txt read hamlet.txt with your stdio.cpp every 4096 bytes.

./eval r f c hamlet.txt read hamlet.txt with your stdio.cpp one by one character.

./eval r f r hamlet.txt read hamlet.txt with your stdio.cpp with random sizes.

./eval w u a test.txt write to test.txt with unix I/O at once.

./eval w u b test.txt write to test.txt with unix I/O every 4096 bytes.

./eval w u c test.txt write to test.txt with unix I/O one by one character.

./eval w u r test.txt  write to test.txt with unix I/O with random sizes.

./eval w f a test.txt write to test.txt with your stdio.cpp at once.

./eval w f b test.txt write to test.txt with your stdio.cpp every 4096 bytes.

./eval w f c test.txt  write to test.txt with your stdio.cpp one by one character.

./eval w f r test.txt write to test.txt with your stdio.cpp with random sizes.

**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\_tests.sh script.

* Execute: ./eval\_tests.sh > eval\_test\_org\_out.txt

7. What to Turn in

|  |  |  |
| --- | --- | --- |
|  | **Criteria** | **Grade** |
| **Code** | **Source code**that adheres good modularization, coding style, and an appropriate amount of comments. Use google coding guidelines: google.github.io/styleguide/cppguide.html | 5 pts |
|  | **Correctness**  **Execution output**from some of the above tests + possibly other non-published tests  •   Correct screenshots of the diff command in Step 4 comparing your output files vs. the originals  diff output\_hamlet.txt Originals/output\_hamlet.txt  diff output\_othello.txt Originals/output\_othello.txt  diff test1.txt Originals/test1.txt  diff test2.txt Originals/test2.txt  diff test3.txt Originals/test3.txt  • Output files from running eval\_tests.sh with your and the original stdio implementations: eval\_test\_out.txt, eval\_test\_org\_out.txt | 20 pts |
| **Report** | **Documentation**of your stdio.cpp implementation including explanations and illustration in one or two pages.  **Discussion**   * Limitation and possible extension of your program * Performance consideration between your own stdio.h and Unix I/O * Performance consideration between your own stdio.h and the Unix-original    stdio.h | 5 pts |
|  | **Total** | 30 pts |

Notes/FAQ

**1) Question**: How are we to handle the myriad of potential errors when implementing these functions? Are we supposed to output specific error codes in line with how <stdio.h> does?

**Answer**: You don’t need to handle all the errors stdio.h does, this is a minimalistic version of stdio.h

**2) Question:** Do we have to implement all buffering modes: \_IONBF, \_IOLBF, \_IOFBF

**Answer:** The majority of the tests will be using \_IOFBF. You do not need to implement \_IOLBF. Please do implement \_IONBF (bypassing buffer) but this will have minimal points attached to it.

**2) Question**: I'm getting all kinds of "redefinition" compiler errors. For example, "Redefinition of printf”. Should I be renaming my stdio.cpp and stdio.h to mystdio.cpp and mystdio.h to avoid these errors?

**Answer**: No, you can just do the following in your stdio.h

#ifndef \_MY\_STDIO\_H\_

#define \_MY\_STDIO\_H\_

And don't forget to include it   #include "stdio.h" // my stdio.h file

**3) Question**: Is there only one buffer for both reading and writing or can we have 2 buffers?

**Answer**: Only one buffer is used

**4) Question**: Are the files test1.txt, test2.txt, test3.txt the same.

**Answer**: Close, the only difference is at the bottom of test3.txt

**5) Question:** What are the common I/O system calls?

**Answer:** The following table shows a summary of common I/O operations open, read, write and close and their corresponding APIs as system calls and C/C++.

|  |
| --- |
| C:\Users\dimpsey\AppData\Local\Microsoft\Windows\INetCache\Content.MSO\F44113F9.tmp |
|  |