CSS 503
Program 3: C/C++ Standard I/O Library

Professor: Munehiro Fukuda
Due date: see the syllabus

1. Purpose

In this programming assignment, we will design our own core input and output functions of the C/C++ standard I/O library, namely stdio.h.

2. Unix I/O

The Unix-based operating systems such as Linux, Mac OS/X, and Solaris provide system calls for file I/O: open( ), read( ), write( ), and lseek( ). However, their problems are two-fold: (1) they are block-based data but not on character-based data transfers, and (2) they are OS-dependent and thus cannot be used in other platforms including Windows. This is the motivation of preparing the C/C++ standard I/O library.

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

It is an architecture independent library that allows C/C++ programs to read and write files without using the underlying OS system calls. The core input and output functions are defined in <stdio.h> and include:

Function name	Descriptioin
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.

For more details, you should use the "man" command that shows the manual page for a given function. Examples: man fopen.

4. FILE Data Structure

Upon a file open, fopen( ) returns a pointer to a FILE object that maintains the attributes of the opened file. The following shows the FILE definition in our own stdio.h.
(It is accessible as ~css503/programming/prog3/stdio.h).

```
#ifndef _MY_STDIO_H_
#define _MY_STDIO_H_

#define BUFSIZ 8192 // default buffer size
#define _IONBF 0    // unbuffered
#define _IOLBF 1    // line buffered
#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; // the actual buffer size when read( ) returns # bytes read smaller than size
  int mode;        // _IONBF, _IOLBF, _IOFBF
  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
```

From the Unix shell, type “man fopen” that shows its specification. The fopen function receives not only a file name to open but also various file access modes:

r Open text file for reading. The stream is positioned at the beginning of the file.

r+ Open for reading and writing. The stream is positioned at the beginning of the file.

w Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

w+ Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

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

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 in accordance with the file modes, allocates a file stream buffer within the FILE object, and actually opens a file using the corresponding OS system call, (e.g., open in Unix).

5. Our stdio.cpp File

In addition to stdio.h, you can also find stdio.cpp in the same directory: ~css503/programming/prog3. This file has already implemented: **printf, setvbuf, setbuf, fopen,** and **feof**. Note that printf accepts only %d, and that the other functions are partially implemented enough to run our driver and performance test programs. You don't have to modify any of them. A user program such as driver.cpp and eval.cpp should include only "stdio.h" but not be aware of the existence of "stdio.cpp". Therefore, "stdio.cpp" was included at the bottom of "stdio.h", in which way you can compile a user program like:

```
g++ driver.cpp
g++ eval.cpp
```

6. Statement of Work

Follow through the six steps show below:

Step 1: Copy the following seven files from ~css503/programming/prog3/ to your directory: compile.sh, driver.cpp, eval.cpp, hamlet.txt, othello.txt, stdio.h, and stdio_template.cpp

Step 2: Change stdio_template.cpp into stdio.cpp, and complete all the implementation of this file.

Step 3: Type "./compile.sh" to compile your program and to obtain driver and eval executables.

Step 4: Run the driver program with "./driver hamlet.txt > output_hamlet.txt", and compare your outputs and ~css503/programming/prog3/output_hamlet.txt, test1.txt, test2.txt, and test3.txt.

Step 5: Run the driver program with "./driver othello.txt > output_othello.txt", and compare your outputs and ~css503/programming/prog3/output_othello.txt, test1.txt, test2.txt, and test3.txt.

Step 6: Run the eval program with the following test cases:

./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 7: 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 evalu program with the following test cases:

./eval r f a hamlet.txt	read hamlet.txt with the unix-original stdio.cpp at once.
./eval r f b hamlet.txt	read hamlet.txt with the unix-original stdio.cpp every 4096 bytes.
./eval r f c hamlet.txt	read hamlet.txt with the unix-original stdio.cpp one by one character.
./eval r f r hamlet.txt	read hamlet.txt with the unix-original stdio.cpp with random sizes.
./eval w f a test.txt	write to test.txt with the unix-original stdio.cpp at once.
./eval w f b test.txt	write to test.txt with the unix-original stdio.cpp every 4096 bytes.
./eval w f c test.txt	write to test.txt with the unix-original stdio.cpp one by one character.
./eval w f r test.txt	write to test.txt with the unix-original stdio.cpp with random sizes.

7. What to Turn in

This programming assignment is due at the beginning of class on the due date. Please turn in the following materials in a hard copy. No email submission is accepted.

Criteria	Grade
Documentation of your stdio.cpp implementation including explanations and illustration in one or two pages. (No more than two, otherwise – 2pts).	20pts
Source code that adheres good modularization, coding style, and an appropriate amount of commends. • 25pts: well-organized and correct code receives • 23pts: messy yet working code or code with minor errors receives • 20pts: code with major bugs or incomplete code receives	25pts
Execution output that verifies the correctness of your implementation and compares your own and the Unix-original stdio.h. • Correct snapshots of the diff command in Step 4 such as diff output_hamlet.txt ~css503/programming/prog3/output_hamlet.txt diff output_test1.txt ~css503/programming/prog3/test1.txt diff output_test2.txt ~css503/programming/prog3/test2.txt diff output_test3.txt ~css503/programming/prog3/test3.txt (+4pts) • A correct snapshot of the diff command in Step 5, namely diff output_othello.txt ~css503/programming/prog3/output_othello.txt (+1pts) • Snapshots of all 16 test cases in Step 6. (+16pts) • Snapshots of all 8 teset cases in Step 7.(+4pts)	25pts
Discussions in one or two pages. (No more than two, otherwise – 2pts) • Limitation and possible extension of your program (+15pts) • Performance consideration between your own stdio.h and Unix I/O (+5pts) • Performance consideration between your own stdio.h and the Unix-original stdio.h (+5pts)	25pts
Lab Session 3 If you have not yet turned in a hard copy of your source code and output or missed this session, please turn in together with program 3.	5pts
Total Note that program 3 takes 11% of your final grade.	100pts