Assignment 8
C/C++ Programming II

Exercise 1 *(10 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add a two new ones, naming them **C2A8E1_OpenFiles.cpp** and **C2A8E1_MergeAndDisplay.cpp**. Also add instructor-supplied source code file **C2A8E1_main-Driver.cpp**. Do not write a `main` function! `main` already exists in the instructor-supplied file and it will use the code you write.

File **C2A8E1_OpenFiles.cpp** must contain a function named `OpenFiles`.
`OpenFiles` syntax:

```
ifstream *OpenFiles(char * const fileNames[], size_t count);
```

Parameters:

`fileNames` – a pointer to the first element in an array representing the names of text files to be opened. The array has the following standard ragged array format:

```
char *fileNames[] = { "fileA", "fileB", etc. };
```

`count` – the number of elements in `fileNames`

Synopsis:

Dynamically creates an array of `ifstream` objects having `count` elements then uses those objects to open the files named in `fileNames`, in order. All opens are in the read-only text mode. If any open fails all previously opened files are explicitly closed, the dynamic allocation is deleted, an error message is output to `cerr`, and the program is terminated with an error exit code. The error message must mention the name of the failing file. If `count` is zero an error message to that effect is output to `cerr` and the program is terminated with an error exit code.

Return:

a pointer to the first entry in the `ifstream` array if `count` is non-zero and all opens succeed; otherwise, the function does not return.

File **C2A8E1_MergeAndDisplay.cpp** must contain a function named `MergeAndDisplay`.
`MergeAndDisplay` syntax:

```
void MergeAndDisplay(ifstream files[], size_t count);
```

Parameters:

`files` – a pointer to the first element in an array of `ifstream` objects having `count` elements, where each object represents a text file open in the text mode for reading.

`count` – the number of elements in the array in `files`

Synopsis:

Proceeding in order from the first file specified in `files`, the first line in each file is read and displayed, followed by the second line in each, followed by the third, etc. When the end of any file is reached that file is closed and the process continues using only the remaining open files until all files have finally been closed. Empty lines are displayed as empty lines. Empty files are simply closed and ignored.

Return:

void

1. Functions `OpenFiles` and `MergeAndDisplay` must be able to handle any number of files specified by their `count` parameter.
2. You may assume that no line in a file will contain more than 511 characters.
3. Do not display anything other than the exact contents of the files, i.e., no file names, line numbers, extra spaces, extra blank lines, etc.
4. Do not attempt to store the entire contents of any file at once.
5. Do not attempt to read a file after reaching its EOF.

6. Do not attempt to use data you "think" you read from a file before verifying that the read was actually successful.

Manually run your program twice, specifying the names of the instructor-supplied data files listed below on the command line in the order shown. Each must be placed in the program's "working directory". DO NOT prompt the user for the file names or place them in your program code.

Command line file names for test 1: **mFile1.txt mFile2.txt mFile3.txt mFile4.txt mFile5.txt**
Command line file names for test 2: **mFile3.txt mFile2.txt mFile1.txt**

Example:

If the command line specifies files **f1 f2 f3** and those files contain the following text:

	f1	f2	f3
Line 1:	`Hello from`	`Bah bah black`	`Now is`
Line 2:	`the`	`sheep`	`the`
Line 3:	`other side of`	`<EOF>`	`time for all`
Line 4:	*`<blank line>`*		`good men and`
Line 5:	`the universe<EOF>`		`<EOF>`

the display would be as follows, where *`<blank line>`* represents an actual blank line:

```
Hello from
Bah bah black
Now is
the
sheep
the
other side of
time for all
<blank line>
good men and
the universe
```

Submitting your solution

Send all three source code files to the Assignment Checker with the subject line **C2A8E1_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

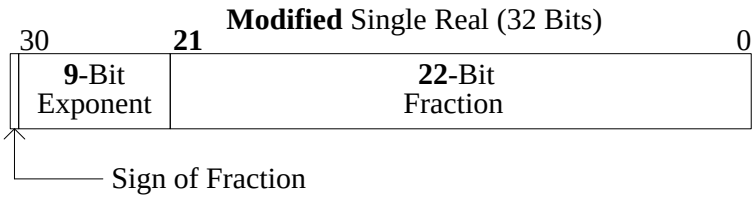
Hints:

C++ does not allow copies to be made of `ifstream` objects. Dynamically create an array of `ifstream` objects with one element for each file to open. These objects may also be used in conjunction with the `is_open` function to determine if a file is still open.

Exercise 2 *(10 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C2A8E2_OpenFileBinary.c** and **C2A8E2_DisplayModifiedSingleReals.c**. Also add instructor-supplied source code file **C2A8E2_main-Driver.c**. Do not write a `main` function! `main` already exists in the instructor-supplied file and it will use the code you write.

A budding young computer scientist working at the local mini-mart has developed a slightly modified version of the IEEE 754 "Single Real" floating point format described in notes 16.10A–16.10C of the course book. The only difference between his version and the original is that the "Exponent" field occupies 9 bits and the "Fraction" field occupies 22 bits. These changes (shown in **bold**) are reflected in the following modified tables:



Modified Single Real			
Normalized Numbers (~limits: ±**1.5*10^{77}**, ±**3.4*10^{-77}**)		Zeros	
Bias of *e*	+**255**	Range of *e*	0
Range of *e*	0 < e < **511**	Mantissa	0.f = 0.0
Range of *f*	Any value	Infinities	
Mantissa	1.f	Range of *e*	**511**
Value	$(-1)^s$ * 1.f * $2^{(e-\mathbf{255})}$	Mantissa	0.f = 0.0
Denormalized Numbers (~limits: ±**8.2*10^{-84}**)		NANs	
Bias of *e*	+**254**	Range of *e*	**511**
Range of *e*	0	Range of *f*	Non-0
Range of *f*	Non-0		
Mantissa	0.f		
Value	$(-1)^s$ * 0.f * $2^{(e-\mathbf{254})}$		

In order to test this modified format two instructor-supplied data files have been supplied, each of which must be placed in the program's "working directory". Each contains an assortment of 32-bit patterns, where these patterns represent various combinations of normalized numbers, denormalized numbers, zeros, infinities, and not-a-numbers (NANs).

File **C2A8E2_OpenFileBinary.c** must contain a function named `OpenFileBinary`.
`OpenFileBinary` syntax:

```
FILE *OpenFileBinary(const char *fileName);
```

Parameter:

`fileName` – a pointer to the name of the file to be opened

Synopsis:

Opens the file named in `fileName` in the read-only binary mode. If the open fails an error message is output to `stderr` and the program is terminated with an error exit code. The error message must mention the name of the failing file.

Return:
A pointer to the FILE structure for the open file if the open succeeds; otherwise, the function does not return.

File **C2A8E2_DisplayModifiedSingleReals.c** must contain function DisplayModifiedSingleReals.
DisplayModifiedSingleReals syntax:

```
void DisplayModifiedSingleReals(FILE *inFile);
```

Parameter:
inFile – a pointer to a FILE structure representing an open readable binary file
Synopsis:
This function assumes a byte is 8 bits and that the file in inFile:

1. was written in "big endian" format;
2. contains successive 32-bit patterns, each representing a "Modified Single Real".

This function displays an aligned table in which each 32-bit pattern is represented by an 8-character hexadecimal value (letters may be uppercase or lowercase) followed by what that value represents if interpreted as a "Modified Single Real". That representation will always be preceded by a plus sign or a minus sign as appropriate. The possible representations are:

1. If a Normalized Number, a Denormalized Number, or a Zero is represented its magnitude will be displayed in scientific notation (using printf's %e conversion specification) followed by the word **Normal**, **Denormal**, or **Zero**, as appropriate;
2. If an infinity is represented **INF** will be displayed;
3. If a not-a-number is represented **NAN** will be displayed.

If the file ends with an incomplete pattern (1, 2, or 3 bytes) the exact message **Unexpected EOF** will be displayed at that point instead of the incomplete pattern.
Return: **void**

1. Do not attempt to obtain a count of the total number of bytes in the file.
2. Do not attempt to read the entire contents of the file at once.
3. Do not make any assumptions about or attempt to determine the machine's "endianness"; a properly written program does not need to know.
4. Do not make any assumptions about the maximum number of bytes in any data types other than the **char** types. For example, you may not assume type **long** has only 4 bytes.
5. Although you may assume a byte is 8 bits, if you actually need to represent that number use the standard library CHAR_BIT macro, not something you define.

Manually run your program twice – once with instructor-supplied input file **TestFile7.bin** and once with instructor-supplied input file **TestFile8.bin**, each of which must be placed in the program's "working directory". Specify the desired file name on the command line – DO NOT prompt the user for it or place it in your code.

Your display must be in the format shown below. This example actually represents the first 12 patterns in file **TestFile7.bin**. The EOF message at the end of the display must occur if and only if a file ends with an incomplete pattern:

```
0xffc00001  -NAN
0x7fffffff  +NAN
0xffc00000  -INF
0x7fc00000  +INF
0xffbfffff  -1.157921e+077  Normal
0x7fbfffff  +1.157921e+077  Normal
0x80400000  -3.454467e-077  Normal
0x00400000  +3.454467e-077  Normal
0x80000001  -8.236092e-084  Denormal
0x003fffff  +3.454467e-077  Denormal
```

```
       0x000d0a00  +7.037971e-078  Denormal
       0x80000000  -0.000000e+000  Zero
       …
       Unexpected EOF
```

Submitting your solution

Send all three source code files to the Assignment Checker with the subject line **C2A8E2_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

Read each 4-byte group from the input file into a 4-element **unsigned char** array and use a 4-iteration loop to place those 4 bytes into the appropriate bytes of a single **unsigned long** variable. Do all necessary masking and testing on that variable.

Get a Consolidated Assignment Report (optional)

If you would like to receive a consolidated report containing the results of the most recent version of each exercise submitted for this assignment, send an empty email to the assignment checker with the subject line **C2A8_*ID***, where ***ID*** is your 9-character UCSD student ID. Inspect the report carefully since it is what I will be grading. You may resubmit exercises and report requests as many times as you wish before the assignment deadline.