Exercise 1 *(6 points – C Program)*

Exclude any existing source code files that may already be in your IDE project, but don't add any new ones. Instead, make a copy of instructor-supplied source code file **C2A7E1_main-Driver.c**, name the copy **C2A7E1_main.c**, and add that copy to the project. This is the only source code file to be used in this exercise.

This exercise is designed not only to familiarize you with binary tree and hashing concepts but to also illustrate the frustration of dealing with someone else's code. The code in the renamed instructor-supplied driver file that you added to your IDE project above represents working but slightly modified versions of both the "binary tree" and the "hashing" code from section 15 of the course book. If macro `TREE` is defined the "binary tree" portion of code will be compiled and if not, the "hashing" portion will be compiled instead. Do the following, in order:

1. Verify that the "binary tree" portion of the code works by ensuring that macro `TREE` is defined then compiling and running the program, noting that the desired input file name must be specified on the command line. Instructor-supplied input file **TestFile1.txt** has been provided for this purpose but you may try any additional text files you wish.

2. Verify that the "hashing" portion of the code works by commenting out the definition of macro `TREE` then compiling and running the program, noting that in addition to the desired input file name the desired number of bins must be specified after it as an additional command line argument. Test with the same input file as above and a bin count of `10`, as well as with any other desired text files and bin counts.

3. Combine, modify, add to, and delete the supplied code in any way you deem necessary so that it will perform the same "hashing" operation as before, but will store the words in ordered binary trees like those in the "binary tree" portion of the code instead of in singly-linked lists. However, to permit automated testing the following two things must not be changed:
    a. The input file name and bin count must still come from the command line.
    b. The display format (spacing, field width, etc.) must be the same as in the original "hashing" version.

If your code is working properly the display it produces will be identical to that of the original "hashing" version except for the order of the words in each bin, which will now be in the order dictated by the standard library `strcmp` function. That is, for input file **TestFile1.txt** and a bin count of `10` the display will start exactly as follows:

```
6 entries for bin 0:
   1  arguments.
   1  constants.
   1  expansion)
   1  invocation
   1  occurrence
   1  parameters
6 entries for bin 1:
   6  a
   1  combination
   1  definition,
   1  definition.
   1  number-sign
   1  stringizing
```

Submitting your solution

Send the modified source code file to the Assignment Checker with the subject line **C2A7E1_*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:

You may use 1 external variable if you wish. I did, but it's certainly not the optimal solution. You may instead implement an additional parameter variable to avoid the external variable if you wish. For my solution I only added about 6 lines of code, deleted about 50 lines of code, and moved a few things around.

Exercise 2 *(4 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add two new ones, naming them **C2A7E2_OpenFileBinary.cpp** and **C2A7E2_ListHex.cpp**. Also add instructor-supplied source code file **C2A7E2_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 **C2A7E2_OpenFileBinary.cpp** must contain a function named `OpenFileBinary`.
`OpenFileBinary` syntax:

```
void OpenFileBinary(const char *fileName, ifstream &inFile);
```

Parameters:

`fileName` – a pointer to the name of the file to be opened
`inFile` – a reference to the `ifstream` object to be used to open the file

Synopsis:

Opens the file named in `fileName` in the read-only binary mode using the `inFile` object. If the open fails an error message is output to `cerr` and the program is terminated with an error exit code.

Return:

void if the open succeeds; otherwise, the function does not return.

File **C2A7E2_ListHex.cpp** must contain a function named `ListHex`.
`ListHex` syntax:

```
void ListHex(ifstream &inFile, int bytesPerLine);
```

Parameters:

`inFile` – a reference to the `ifstream` object for a file that is open in a readable binary mode
`bytesPerLine` – the number of bytes to displayed on each line as long as bytes are available

Synopsis:

Displays the contents of the file in `inFile` as one pair of hexadecimal characters per file byte, zero-filled on the left if necessary to produce the two characters. Pairs are single-space-separated and the number of pairs that are placed on each line is specified by parameter `bytesPerLine`. Pairs are aligned from one line to the next and the last line will contain less than the specified number of pairs per line if EOF is reached prior to the completion of that line. You may assume that a byte consists of 8 bits for this exercise.

Return:

void

Typical output from `ListHex` with 16 bytes per line must look like:

```
00 AB 05 72 FE 01 03 67 68 69 20 40 78 0A 0D 02
AB 05 72 FE 01 AB 05 72 FE 01 20 40 78 67 68 69
FE 01 03 67 68 69 20
```

The instructor-supplied driver file that calls `ListHex` requires that you provide the desired input file name followed by the number of bytes per line as command line arguments.

Manually re-run your program several times, testing with at least instructor-supplied files **TestFile3.txt** and **TestFile4.bin** and several different bytes-per-line values.

Submitting your solution

Send all three source code files to the Assignment Checker with the subject line **C2A7E2_*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:

1. Display the value of each and every byte in the file but do not display the values of any bytes that are not actually in the file.

2. Do not display EOF since it is not a character in the file and cannot be displayed meaningfully.

3. Use the `setw` and `setfill` I/O manipulators to obtain the correct field widths and fill characters, respectively. `setw` applies only to the next item to be output whereas `setfill` remains in effect until explicitly called again. `setw` is discussed in the course book and information on both is readily available in any C++ text or online.

4. Due to "sign extension" bytes whose values are from `0x80` through `0xFF` may be printed with unwanted leading **F**s unless they are stored in or typecast to **unsigned char** before printing. Bitwise ANDing the values with `0xFF` prior to printing is another, but less efficient, way to avoid this problem.

Exercise 3 *(4 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C2A7E3_ReverseEndian.c**. Also add instructor-supplied source code file **C2A7E3_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.

File **C2A7E3_ReverseEndian.c** must contain a function named `ReverseEndian`.

`ReverseEndian` syntax:

```
void *ReverseEndian(void *ptr, size_t size);
```

Parameters:

`ptr` – a pointer to the object whose endianness is to be reversed

`size` – the number of bytes in the object

Synopsis:

Swaps the bytes in the object in `ptr`, thereby converting it from big endian to little endian or vice versa. `ReverseEndian` will fail if the object is not scalar or contains padding.

Return:

`ptr`

Submitting your solution

Send both source code files to the Assignment Checker with the subject line **C2A7E3_*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:

None

Exercise 4 *(6 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add three new ones, naming them **C2A7E4_ReverseEndian.c**, **C2A7E4_ProcessStructures.c**, and **C2A7E4_OpenTemporaryFile.c**. Also add instructor-supplied source code files **C2A7E4_Test-Driver.h** and **C2A7E4_main-Driver.c**. Do not write a `main` function! `main` already exists in the instructor-supplied implementation file and it will use the code you write.

The purpose of this exercise is to illustrate the endian conversion of arbitrary scalar objects within an aggregate type, such as a structure or class. If it is arbitrarily assumed that a **long** is 4 bytes and a **short** is 2 bytes, a structure defined and initialized as

```
struct { long height; short width, depth; } box = {0x01234567L, 0x89ab, 0xcdef};
```

and written into a file using

```
fwrite(&box, sizeof(box), 1, fp);
```

will be written exactly how it appears in memory, which is in one of the two following orders, depending upon the machine's endianness. There can be an arbitrary amount of implementation-dependent padding after any members:

	Big Endian	Little Endian
First byte in memory & in the file:	01	67
	23	45
	45	23
	67	01
	----- possible padding here -----	
	89	ab
	ab	89
	----- possible padding here -----	
	cd	ef
	ef	cd
	----- possible padding here -----	

Regarding data type **`struct Test`**, which is used in this exercise…

`struct Test` is a data type that is defined in instructor-supplied header file **C2A7E4_Test-Driver.h**

This header file must be included (**`#include`**) in any file that uses this data type.

File **C2A7E4_ReverseEndian.c** must contain a copy of the `ReverseEndian` function you wrote for the previous exercise, modified if necessary for this exercise.

File **C2A7E4_OpenTemporaryFile.c** must contain a function named `OpenTemporaryFile`.

`OpenTemporaryFile` syntax:

```
FILE *OpenTemporaryFile(void);
```

Parameters:

none

Synopsis:

Opens a temporary file using the standard library **`tmpfile`** function. If the open fails an error message is output to **`stderr`** and the program is terminated with an error exit code.

Return:

A pointer to the open file if the open succeeds; otherwise, the function does not return.

File **C2A7E4_ProcessStructures.c** must contain functions named `ReverseMembersEndian`, `ReadStructures`, and `WriteStructures`.

`ReverseMembersEndian` syntax:

```
struct Test *ReverseMembersEndian(struct Test *ptr);
```

Parameters:

`ptr` – a pointer to the structure whose members' endiannesses are to be reversed

Synopsis:

Calls function `ReverseEndian` for each member of the structure in `ptr`, thereby converting each from big endian to little endian format or vice versa. `ReverseEndian` will fail if the member is not scalar or contains padding.

Return:

`ptr`

`ReadStructures` syntax:

```
struct Test *ReadStructures(struct Test *ptr, size_t count, FILE *fp);
```

Parameters:

`ptr` – a pointer to where the structure(s) that are read will be placed in memory

`count` – the number of structures to read

`fp` – a pointer to an open binary file containing the structure(s) to be read.

Synopsis:

Reads `count` structures from `fp` and stores them in memory starting at address `ptr`. If `count` structures can't be read an error message is output to `stderr` and the program is terminated with an error exit code.

Return:

`ptr` if `count` structures are read; otherwise, the function does not return.

`WriteStructures` syntax:

```
struct Test *WriteStructures(const struct Test *ptr, size_t count, FILE *fp);
```

Parameters:

`ptr` – a pointer to where the structure(s) that are written will come from in memory

`count` – the number of structures to write

`fp` – a pointer to an open binary file into which the structure(s) will be written.

Synopsis:

Reads `count` structures from memory starting at address `ptr` and writes them to the file `fp`. If `count` structures can't be written an error message is output to `stderr` and the program is terminated with an error exit code.

Return:

`ptr` if `count` structures are written; otherwise, the function does not return.

- Do not use dynamic memory allocation.
- Do not make any assumptions about the number of bytes in any data type.
- Do not make any assumptions about the presence, absence, value, or amount of padding.
- Please answer the following questions about your results and place these answers as comments in the title block of the file containing the `ReverseMembersEndian` function:
    1. Were the results you got correct for your implementation?
    2. How many padding bytes were in your structure?

Submitting your solution

Send all four source code files to the Assignment Checker with the subject line **C2A7E4_*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:

Having trouble getting the `tmpfile` function to work?

If you are running Windows Vista or newer the failure of **`tmpfile`** is probably because your program lacks administrative privileges. It is important to realize that being logged onto the computer with administrative privileges does not automatically bestow those privileges on programs you run. Instead, it merely permits you to bestow them. If you are running your program from within the Visual Studio IDE it will not have administrative privileges unless Visual Studio itself does. To accomplish this start Visual Studio by right-clicking its icon or Start Menu item and selecting "Run as administrator". If you are running your executable program directly from its own icon or .exe file, right-click that icon or **`.exe`** file and select "Run as administrator".

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 **C2A7_*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.