COMPUTER SCIENCE 241
Spring 2023

Programming Assignment 8

Due by end-of-day on Monday, May 8

For this assignment, you will use the C functions fopen, fclose, fseek, ftell, fread, fwrite, feof, etc. to do file processing. The input to your program will be a file full of records containing some sort of information. Your program's task is alter the contents of the file so that the records are now stored in **reverse order.** The contents of each individual record are still presented in the same order as before, but the order of the records themselves is reversed. For example, recall the simple "database" of Person structs that we considered in the class lessons.

```
$  cat  db.txt

ann F 1998

bob M 2000

sue F 2003

tom M 1997
```

Of course, this file is really stored as a binary file (not a text file) as shown below.

```
$ od  -An  -x  -w8  --endian=big   db.bin

616e 6e00 4600 ce07

626f 6200 4d00 d007

7375 6500 4600 d307

746f 6d00 4d00 cd07
```

This file **db.bin** contains four records, and each record has a size of eight bytes. Executing your program should alter the file db.bin, whose contents should now be:

```
$ od  -An  -x  -w8  --endian=big   db.bin

746f 6d00 4d00 cd07

7375 6500 4600 d307

626f 6200 4d00 d007

616e 6e00 4600 ce07
```

Notice that this file contains no newline characters. Each record appears on a separate line above only because the –w8 option was used in the od command.

Note also that the file may have any number of records, unlike the code presented in the class notes, which assumed that the db.bin file contained exactly four records. Your program should work for a file with any number of records (including zero!).

To accomplish this task, your program will read data (using fread) into main memory (RAM), and then write that data back out to the file in the correct location in that file. Note that your program is altering the **original** file and not creating a second file. In other words, your program ism manipulating the data **"in place".**

Files are stored on secondary memory (aka storage, e.g., hard disk drive). Any read/write to storage is very slow, compared to manipulating variables in main memory. Since each file read is slow, the OS optimizes system performance by reading an entire **page** from storage. For example, suppose you want to eat a sandwich for lunch. You could drive to the grocery store, buy a loaf of bread, and then drive home. You could then drive to the grocery store again, buy some mustard, and then drive home. You could then drive to the grocery store again, buy some ham, and then drive home. You could then make your sandwich and eat it. Clearly, it would be more efficient to make only one trip to the grocery store. The same idea applies to accessing data in storage. You can execute

```
char foo;

fread ( &foo, 1, 1, someFilePtr );
```

to read a single char/byte into the variable foo. But since you are going all the way out to storage anyway, you might as well read an entire **page** of data. For example, the page size on our Unix host is 4,096 bytes.

Your program will take the following input on the command line.

- argument argv[1] indicates the size (number of bytes) in each record in the file (e.g., 8 bytes for the simple db.bin file)
- argument argv[2] indicates the page size, in bytes, for the system you are using
- argument argv[3] is the name of the file that you are altering

For example, invoking your program might look like:

```
$ ./a.out   20   1000   myInputFile
```

In this example, each record in myInputFile file contains 20 bytes, and each page can hold 1,000 bytes. The size of the entire file is some much larger number (perhaps 2,000,000,000 byes). Therefore, each fread operation should fetch in 50 records. For simplicity, we will assume that the record size evenly divides the page size. Of course, the record size evenly divides the file size, since the file is simply a sequence of records. But you should **not** assume that the page size evenly divides the file size. Your program must **minimize** the number of reads/writes to the file system by fetching data in pages, instead of one record at a time.

Your program **must** only use O(page size) amount of RAM. You **must not** attempt to read the entire file into RAM, reverse the file in RAM, then write the entire file back out to storage. This is not feasible because our file is so very large, it cannot fit inside of RAM.

Every argument on the command line is a string. You can convert a string into an integer using the "ASCII to Integer" function **atoi** in **stdlib.h**, as follows:

```
int  foo =  atoi( argv[1] );
```

Consider the following example input file called **myInput**:

```
ABCdefGHIjklMNOpqrSTUvwx
```

This file contains exactly 24 bytes (no newline chars).

If your program is invoked as:

```
$ ./a.out   3   6   myInput
```

Then the resulting output should be the 24 bytes:

```
vwxSTUpqrMNOjklGHIdefABC
```

If your program is invoked as:

```
$ ./a.out   3   12   myInput
```

Then the resulting output should be the **same** 24 bytes shown below (since the page size has no effect on the final output; the page size only impacts the efficiency of the program's execution).

```
vwxSTUpqrMNOjklGHIdefABC
```

If your program is invoked as:

```
$ ./a.out   6   12   myInput
```

Then the resulting output should be the 24 bytes:

```
STUvwxMNOpqrGHIjklABCdef
```

In this tiny example file **myInput**, the "record" is very small and truly meaningless. But C sees every file as simply a stream of bytes. Whether or not those bytes make any sense is completely irrelevant to the C program.

Sample test files are provided in the handouts folder. You can copy these files using the commands:

```
$cp    ~joanmlucas/cs241/programs/prog08/handouts/input*        .

$cp    ~joanmlucas/cs241/programs/prog08/handouts/output*      .
```

There is one test input file called **inputChars_3**. This file assumes a record size of 3 bytes. The correct file that results from reversing **inputChars_3** is the file called **outputChars_3.**

There is one test input file called **inputSmall**, with file size of 24 bytes. The correct file that results from reversing inputSmall, assuming a record size of 6 bytes, is the file **outputSmall_6.** The correct file that results from reversing inputSmall, assuming a record size of 8 bytes, is the file **outputSmall_8.** The correct file that results from reversing inputSmall, assuming a record size of 12 bytes, is the file **outputSmall_12.**

There is one test input file called **inputBig**. The correct file that results from reversing inputBig, assuming a record size of 4 bytes, is the file **outputBig_4.** The correct file that results from reversing inputBig, assuming a record size of 20 bytes, is the file **outputBig_20.** The correct file that results from reversing inputBig, assuming a record size of 250 bytes, is the file **outputBig_250.**

You can test whether the resulting file produced by your program matches the correct answer using the diff command. The command:

```
$ diff someFile anotherFile
```

will produce no output at all if the two files are identical.

Notice that the "small" and "big" files are not text files. You should use the od command (octal dump) if you want to examine the contents of these files. Since we are only looking at individual bytes here, the "endian" does not matter, and you can ignore that option. Note also that when you generate the output file, the result should be **same** no matter what (legal) page size you use.

Your program **must not** use any storage beyond the original file. In other words, your program must operate "in place" with respect to storage. Recall that Bubblesort and Heapsort are both "in place" algorithms, but Merge Sort is not.

Your program should return an exit code of 0 if it successfully completes the reversal. Otherwise your program should return an exit code of 1. Possible error modes are: the specified file does not exist, the record size does not divide the page size, etc.

You should submit your program for grading by copying it into my account using the following command (assuming your name is J. Smith)

```
$ cp   smith_j_prog08.c    ~joanmlucas/cs241/programs/prog08/submission
```

The timestamp of this copied file will indicate when you submitted this assignment. **After** you have copied the file, you must **be sure to set the permissions** on this copied file as follows:

```
$ chmod  644  ~joanmlucas/cs241/programs/prog08/submission/smith_j_prog08.c
```

You can verify that your file was successfully copied using the ls (list) command:

```
$ ls -l  ~joanmlucas/cs241/programs/prog08/submission/smith_j_prog08.c
```