CMPE-380: Applied Programming

Laboratory Exercise 12

Input/Output Reading & Sorting

Table of Contents

In this lab session, we will review the general input, output and sorting features in the C language. In general, the input features of C are weak and can cause application errors in the real world but the sorting function is very powerful.

Pre-Lab – 10 pts

In this lab session, we will examine the sort function and simple command line input. Read and understand the following sections. Your TA will conduct a brief oral quiz, in addition there are also some programming assignments in the pre-lab.

Reading and writing files

System hard disk files are read and written using file handles. Files are opened using fopen(), closed using fclose() and then read/written using various functions. The standard Linux file handles (stdin, stdout & stderr) are always open in a C program and can also be used as "files".

All the standard file functions are defined in "**stdio.h**". All file operations use a file handle structure called "**FILE ***" (all caps).

The standard file access functions are:

```
FILE *fopen()   — open file
int fclose()    — close file
int fseek()     — move the file pointer
int fflush()    — flush output buffer
int feof()      — check end-of-file
```

Functions in the standard library that use a pointer to FILE are ***usually buffered***. This buffering feature improves overall I/O performance but can result in loss of data in the case of an OS crash and can also generate confusion when debugging code. You might see a "print" in your debugger code but then not see the output on the screen.

fflush() can be use on any file handle, including stdout/stdererr, to force the OS to write the data out.

e.g. fprintf(stderr,"Message I want\n");
fflush(stderr);

fopen()

The fopen() command opens a file and returns a "file handle", or "handle" to the file. Syntax is:

FILE *file_p = fopen("file name", "file mode");

Files can be opened in the following mode:

r	read text file (r+ , read & write)
w	write text file (w+, write & read)
a	append to text file
rb	read binary file
wb	write binary file
ab	append to binary file

The C language makes a distinction between "binary" and "text" modes to support the differences between Linux "\n" and Window "\n\r" text termination. The default text mode ensures compatibly between these two termination types. "Binary" mode MUST be use whenever processing binary data to prevent accidental "\r" character conversions.

Note 1: If you forget to fclose a FILE* you get a memory leak AND you could run out of file handles. Some OS versions may have a limited number of file handles. It is always a good idea to close files as soon as you can to save system resources.

Note 2: Opening a file that is already open in your code has unpredictable results, only keep one open file handle for each file you are processing.

fread()

fread() is used to read from an input handle into a buffer:

nRead = fread(void *buffer, int esize, int elem, FILE *handle);

Reads "elem * esize" bytes. Hint: make esize '1' so you read bytes
buffer must be at least "esize*elem" bytes

nRead returns the number of objects read (or bytes if esize is 1)

If nRead is less than your buffer size, you know you got all the data. If nRead equals your buffer size then you don't know if you got all the data. Always make your read buffers a little larger then you need.

E.g.: read up to 255 bytes from handle and put it in buffer

```
num = fread(buff, 1, 255, handle);
```

fseek()

The fseek() command allows us to "move around" in a file. You can use the command to move the file r/w head to the start, end or anywhere inside the file. This can be useful when creating "rolling" logging buffers or for performance measurements.

int = fseek(FILE *handle, long int offset, int whence);

Whence values:

SEEK_SET	Beginning file
SEEK_CUR	Current position in file
SEEK_END	End of file

E.g.: re-start reading or writing at the beginning of a file

```
rc = fseek(handle, 0, SEEK_SET);
```

ftell()

The ftel() function returns the current value of the file pointer, this, in combination with fseek() is useful when you wish to re-read the same section of a file.

long int ftell(FILE *stream)

stat()

The stat() function returns information about a file, like its' existence, file size, creation date and R/W status. This function requires the additional include file "**sys/stat.h**"

```
int = stat(const char *filename, struct stat *stats);
char *filename        The name of the file to check, note NOT A FILE HANDLE
struct stat stats     Pointer to a stat data structure to hold the results

Returns "0" for success or specific errors
```

e.g. a quick check to see if a file exists and the file size:

```
struct stat file_status;
if (stat("file.txt", &file_status) != 0) {
  fprintf (stderr,"File not found\n",);
}
else {
  Printf("size %d\n", status.st_size);
}
```

scanf()

The scanf() family of functions are used to parse simple fixed format data. The family includes:

scanf() — from standard input (file stdin)

fscanf() — from any file

sscanf() — from a string

The scanf() family use the printf() family of format commands to parse data:

%d: int

%ld: long int

%u: unsigned int

%lu: unsigned long int

%c: char

%s: string (i.e., *pointer to char array*)

%f: float or double (*)

%x: hexadecimal, using lowercase letters

%X: Hexadecimal, using uppercase letters

Like printf(), scanf() functions can take a length identifier in the embedded format string. This is very useful when reading strings as it will prevent buffer overflow. E.g. **%10s** will limit the parsed input string to 10 characters (or a buffer size of 11). The following code fragment can be used to dynamically create parsing based on #define macro sizes

```
sprintf(parseCMD, "%%%ds", MAX_STR);       // e.g  “%255s”
```

The scanf() functions use the format string to control data conversion and require the ADDRESS of the variable(s) to store the converted results. The scanf() functions return the number of objects converted, **EOF** or **-1 for error**. You should always check the number to ensure scanf() was able to convert the data. The prototype is: **int scanf(const char *format, ...)**

For example, to parse a string as a number from the command line use:

```
float miles;
rc = scanf("%f", &miles);       // be sure to check the return code from scanf()
```

Using the provided **lab_a.c** template, write a program to use scanf() to convert miles data from the command line and convert it to kilometers. Continue the conversion while the input is positive, non zero. Do NOT check the return code from scanf().

1) Test your program with each of the values: **1** **2.2** and **your first name**.

Describe what happened for each test case in your **prelab.txt** file, did the last do what you expected?

Now add error checking to the scanf() function. Print out the following message "**Error, nothing parsed**" and exit with a non-zero return code if the number of objects converted is zero.

1) Retest your program with **1** **2.2** and **your first name**

Describe what happened for each test case in your **prelab.txt** file, did the last test case do what you expected?

fgets()

The scanf() family of function are very useful when the input data is known good but it is not very robust when the input data is possible incorrect, normally due to human editing. The fgets() function, along the strtok() function give us full access to the input parameters.

The fgets() function is a text line oriented read function. It will read up the buffer size limit (n-1) OR until the first newline character and then insert a NULL character to terminate the string. It returns a pointer to the supplied buffer or NUL of no data was read.

char *fgets(char *line, int n, FILE *fp);
line - pointer to the read buffer
n - size of the read buffer
fp - pointer to an input handle

Returns- pointer to “line” or NULL for empty

Note 1: fgets()will read up to n-1 characters from fp, it will NUL terminate as soon as a newline or end-of-file is encountered.

Note 2: fgets() returns the new line “\n” character so it is often a good idea to replace the new line with a NUL so the string is “normal”. This is application specific.

strtok()

strtok() is often used with fgets() as strtok() will incrementally parse a buffer and return pointers to each parsed segment. Often, it is coded in a while loop to handle an unknown, variable number of white space delimited input parameters. strtok() has two weaknesses:

1) Strtok() can't identify trailing white space and will count the trailing white space as the value "zero". Due to this weakness, it is important to always truncate the output from fgets() so there are no trailing white spaces. White space characters can be " " \n, \r and \t.

2) Strtok() is destructive so the input buffer is altered. This weakness is normally not an issue because once you parse a parameter you don't need that buffer segment anymore. If the destruction is an issue, make a copy of the buffer before using it.

char *strtok(char * s1, const char *s2);

parses a string like scanf(), must be called iteratively
s1 - first call, the *string to be searched, subsequent calls NULL, s1 is destroyed*
s2 - argument is string of *token separators*
Returns – NULL when finished

How it works: sttok() searches **s1** using parse characters in **s2.** Often s2 is just " \t" (parse on tabs or spaces). If s1 contains one or more tokens, the char following the token is overwritten with a null character and a pointer to the 1st character in the token is returned. Subsequent calls with s1 equal to NULL return a pointer to the next token, etc. If no additional tokens are available, NULL is returned. Typical usage:

```
// Use to access the parsed parameter
char *p;

// begin parsing
p = strtok(buffer," \t");

// Parse until no tokens left
while ( p != NULL ) {
   printf("\"%s ", p);

   // Parse the next token, returns NULL for none
   p = strtok(NULL, sep);
}
```

Implementation hint:
Sometimes you must allocate data arrays, but you don't know how much data you have until you parse the entire input line. You could dynamically allocate each element as you go along (a pain) or you could make a copy of the input line, use strtok() to quickly count the number of parameters, malloc the require memory and then really parse and convert the data using strtok() on the copy buffer. Remember, strtok() is destructive.

e.g.

strcpy() the input data to make a 2nd copy
strtok() the 2nd copy to count the entries
allocate the data space
strtok() the original input data

When strtok() is used with white space (" ") parsing, any trailing spaces are reported as a token. The solution to this issue is to remove trailing spaces in your buffer before using strtok().

```
i = strlen(data)-1;                  /* C is origin 0 */
/* Make sure we don't go negative */
while ((i >= 0) && (data [i] == ' ')) {
    data [i] = 0;
    i--;
  }
printf("'%s'\n", data);    /* Now data has no trailing spaces */
```

Note: This style of code is useful to remove other "bad" things like tab characters:

```
if (data [i] == 0x09) {data [i] = ' ';}    /* 09 HEX is TAB */
```

or other characters like \n and \r.

Command Line input:

Most command line parsing in this class used get_opt() and get_opt_long_only(). In this section we will examine the traditional command line parsing support.

"C" has a standard entry point called "main" which provides the ability to pass **N string** arguments. The operating system will take each space delimited parameter on the command line and assign it to an argv[] entry. The total count of items (origin 1) will be placed in argc.

The operating system will also parse each operating system environment variable and assign them to env[] but the operating system does not provide a count. The last env[] pointer will be NULL indicating the end of the list.

Prototype:

```
int main(int argc, char *argv[], char*env[])
```

argc – the number of arguments on the line

argv – pointer to the read only argument strings

env – pointer to the read only environment strings, last element is NULL

There will always be at least ONE parameter on non-embedded systems, arg[0] contains the name of the program.

Use the provided **lab_b.c** as a template and write code to print out all the command line arguments and environment variables, clearly indicating which element is being printed. Run your program with the following and redirect the output to your **prelab.txt** file.

```
./lab_b hello world
```

Qsort

Qsort is a general purpose, linear array sorting function that can sort any 1D array of data using a programmer defined comparison function. The function is defined in **stdlib.h**

void qsort(void * array, size_t n_els, size_t el_size, int (*compare)(const void *, const void *));

array	- array to be sorted
n_els	- number of elements in the array
el_size	- size (in bytes) of each array element
compare	- pointer to a programmer supplied compare function

Qsort calls the programmer defined compare function with pointers to TWO elements to compare and the function must then return:

-1 - p1 goes before p2

+1 - p2 goes before p1

0 – if the elements are equivalent.

The following is the compare function protype:

int compare(const void *p1, const void *p2);

The data elements pointed to by p1 and p2 can be base types like integers and floats or can be user defined data structures. The function must return **-1, 0 or +1** (See above)

Update the provided **lab_c.c** program to take ONE parameter, the number of entries to create and sort the data in numerical order. The minimum value is two elements. You must use traditional command line parsing and handle all errors, do not use get_opt() or get_opt_long(). Return a non-zero value for error and zero for success. The atoi() function can be used to convert a string to an integer.

Run your program with the following and redirect the output to your **prelab.txt** file.

./lab_c

./ lab_c -1

./ lab_c 6

./ lab_c 1 2

Your TA will conduct a brief oral quiz, verify your **lab_a.c, lab_b and lab_c.c** file and verify the results of your **prelab.txt** file

Interactive Examples – 30 pts

In this

Qsort:

You will expand your Qsort prelab to sort a more complex data type. Sort the polar notation data by magnitude first and angle second. Use the provided **lab_d.c** as a frame work.

Run the following examples and save the results in **exercise.txt**

./lab_d

./ lab_d -1

./ lab_d 6

./ lab_d 1 2

Scanf():

Reading string input data using of the scanf() family functions is not secure if a simple "**%s**" format string is used. This is because strings must be read into buffers and buffers have limits. We need a way to control the string read size and the scanf() family provides length control with the "length format" option. If you include a length field in the simple format string (e.g. "**%5s**") you can control the number of bytes read into the buffer and therefore prevent overflows.

The provided **lab_e.c** file reads a string from the keyboard and prints the string and string length out. **memtst** is a valgrind script that can be used to verify **lab_e**.

1) Read and understand the lab_e.c code
2) Build the lab_e.c code
3) Run the **./memtst ./lab_e** script with the following four input values:
   "**1234567**" "**12345678**" "**123456789**" and "**q**"

4) Why is "string" malloced with MAX_STR+1 and not just MAX_STR? What output values are printed for each step? Does valgrind report any errors? Why? Document your answers in **exercise.txt**

Modify the code to automatically insert the macro value MAX_STR into the format string. Use the sprintf() command to dynamically build the format string.

1) Re-build the lab_e.c code
2) Run the **./memtst ./lab_e** script with the following four input values:
   "**1234567**" "**12345678**" "**123456789**" and **"q"**
3) What output values are printed for each step? Does valgrind report any errors? Why not? Is there anything "funny" about the output now? Document your answer in **exercise.txt**

Re-compile your code with **-DMAX_STR=7**, you may NOT modify the source code in any other way.

1) Run the **./memtst ./lab_e** script with the following four input values:
   "**1234567**" "**12345678**" "**123456789**" and **"q"**
2) What output values are printed for each step? Does valgrind report any errors? Why not? Is there anything "different" about the output now? Document your answer in **exercise.txt**

fgets():

In this section we will use fgets(), fscanf() and fseek() to illustrate some of the weakness in the scanf() family of parsers. You are provided with data.txt, a data file that includes extra embedded space, trailing spaces, leading spaces and input data with either too much data or too little. In this exercise you will see that the scanf() family of parsing routines is only suited for "good" data.

Use the provided **lab_f.c** file as a template for your work. Review the code to determine the tasks required. The input data is assumed to be of the form" integer integer integer", you will use fgets() to read the actual data and print it out so you can see the leading and trailing spaces. You will need to remove embedded LF characters in your buffer before printing the output. Your output should be:

```
Read: '1 2 3'
Read: '4    5 6    7     '
Read: '8 9'
Read: ''
Read: ''
Read: ''
```

You will then use the fseek() function to reset the file pointer to the beginning of the file so you can then re-read the data using fscanf(). Assume you are reading 3 integers and that you don't have any knowledge of the "bad" input data. Use the provided **memtst** script to run valgrind on your binary and resolve all memory and access errors.

1) Did fscanf() produce the data you expected and wanted? Is the data really correct? Document your answers in you **exercise.txt** file
2) Include the valgrind error report in your **exercise.txt** file, use:

```
./memtst lab_f > mem.txt 2>&1
cat mem.txt >> exercise.txt
```

Assignment: Data Parsing - 60 pts

In this section you are going to build a robust, dynamic data parser to read in linear matrix data and then use the GSL LU matrix solver tool to generate the solution. The input matrix data can be of any size, you must dynamically grow your read buffers to accommodate the input data. Your solution can not read the entire input file before processing the data, your code must dynamically adapt. Your program many not use "excessive" buffers.

Standard mathematical matrix data is in the form:

Ax=b where A is the 2D matrix of coefficients of the unknows in vector x, and b is a vector of given values.

Computer engineers store this Ax=b this data in "augmented form" which is :

M = [A b] Where M is the matrix data made up of A, the 2D matrix of coefficients and b, vector of given values.

The data provided in this section of the lab will have the following form:

1) Start with zero or more comment lines. The comment character is "#"
2) The first non-comment line documents the expected size of the augmented matrix, rows and columns,
3) The number of columns must always be one more than the number of rows.
4) There must be a minimum of 2 rows.
5) The Remainer of the lines in the file represent the "M" matrix which is made up of "A" matrix values and "b" vector values.
6) Actual data may contain embedded tabs, trailing tabs, white space, CR & LF characters.
7) Some data will be incorrect and have missing or extra data elements.
8) There is no need to support embedded comments. Comments are only allowed in the starting lines of the data file.

The following is a typical data file:

```
# Matrix with leading space, trailing whitespace, trailing \n and \r, good matrix size
# three lines of comments, Solution -1.7500,  -1.0000, 5.5000
# The M matrix data should have 3 rows and 4 columns.
 3 4
 2 1 1  1
 6 2 1 -7
-2 2 1  7
```

The "A" matrix data would be:

```
 2 1 1
 6 2 1
-2 2 1
```

And the "b" vector data would be:

1
7
7

In your implementation expect to use the following functions:

fseek()
ftell()
fgets()
strtok()
realloc()
GSL_MAT_ROWS()
GSL_MAT_COLS()
gsl_matrix_calloc();
gsl_vector_set();
gsl_matrix_set();

The "main()" function with the complete GSL calculation suite is provided in the **hw12.c** template. Your program should return errors using the values in **ClassErrors.h**. There are thirteen test files provided: ge0.txt … get12.txt and rand.txt. Be sure to review the comments in the data files.

This program uses GSL so your link line will require: **-lgsl -lgslcblas**

Note: Expected results have been included. The program **genRand** has been included, it should be used to build the 2K random data "**rand.txt**"

Makefile

You must provide a quality Makefile with the following targets: all, tests, mem, clean and help.

"all" -should make **Hw12**

"tests" - should run the ge0.txt … ge12.txt cases and redirect the output to out.txt. Application errors should not stop the make file.

"mem" -should run rand.txt and redirect output to mem.txt.
It should build rand.txt from genRand when necessary

help, clean - should do the normal things

Analysis

No **analysis.txt** file is required for this assignment

Deliverables

All the files you create to accomplish this task should be packed in a single tar file **lastName_Hw12.tar** (lastName is your last name). In addition to your C source code, you should ***include all files*** *you created .* **Please do NOT include ran.txt in your tar file, it is 40MB.**

Grading Criteria

1. (39 points) 13 * 3pts each, ge0.txt to ge12.txt.
2. (11 points) Memory leaks in rand.txt
3. (10 points) Makefile

Student Name: _____

Laboratory Grading Sheet

Lab 12 - Input/Output Reading and Sorting

Component	Point Value	Points Earned	Comments and Signatures
Pre-Lab: lab code and results: a/b/c,	9		
Pre-Lab: quiz	1		
Interactive Exercises: qsort lab_d	10		
Interactive Exercises: scanf() lab_e	10		
Interactive Exercises: gets() lab_f	10		
Total	40		

You must turn this signed sheet in at the end of lab to receive credit!