CS 420
PROGRAM 4P

Purpose

The purpose of these programs is to gain familiarity with reading information from different types of files, and processing them based on their specifications. It also gives you practice to write your own test drivers with main() functions for each.

Along with the C source files you must also provide **one makefile** which has targets that builds object files required for both programs as well as a target called "**clean**" that removes both all object files.

Files to be submitted: **calSpan.c, dataDump.c, makefile**

Program 1

Files to be submitted: **calSpan.c, (common) makefile**

There is a file called calSpan.h in the public folder which shows the signature of your function calSpan(). You do not submit this header file.

Suppose a temperature sensor collects data once per minute. The sensor data could be weather data, body temperature, etc. The data are real numbers (floating-point values), in the range of **[-100, 100]**. Any values outside this range are considered invalid/corrupt. A value of 0 is also considered invalid/corrupt. A *high temperature span* for minute *n* is the number of minutes prior to minute *n* such that the temperature values for those minutes were are all lower than or equal to the value at minute *n*.

For example, 10 temperature entries are shown; values start from minute 0, 1 and so on.

37.8	38.9	39.1	40.5	37.9	38.8	40.5	39.0	36.9	39.8
0	1	2	3	4	5	6	7	8	9

At minute 2, the corresponding data is 39.1 and it is a 3-minute high. At minute 6, the corresponding data is 40.5 and it is a 7-minute high.

You are asked to write a function called `calSpan()` to read from a file that stores sensor data, and to compute spans based on input given by the user—the user will indicate a minute of interest and the program will tell the user how long the span is for that minute. All sensor data are stored as space-delimited ASCII strings in the data files. Users type in queries at keyboard to find out the data span at any given minute. The program provides answers in a format like this:

```
Data at minute 0 is a 1-minute(s) high.
```

The function should open the filename specified in an input argument, read data and appropriately update the array supplied as an argument (restricting the number of data items read to the maximum size specified as an argument) and do the following: It should eventually return the actual number of data items processed by the function (which would be any number less than or equal to the maximum specified size), prompt the user to input the index of a minute by displaying "**`Which minute to query?`**" The program should support repetitive user query until the user inputs the word `exit` (case insensitive) with a newline, at which point the function terminates and returns to the main() function.

You will not be submitting any file that contains a main() function that's needed to run your calSpan() function. But you will need to test your function by writing a main() function of your own. Keep in mind that you will have to write your own main() function in a separate file. Shown below is a sample run of your function when invoked from a main() function that supplies the correct arguments.

Let’s say this is the content of your data file (a few data files are available in the public folder):

```
37.8 38.9 39.1 40.5 37.9 38.8 40.5 39.0 36.9 39.8 37.8 41.1
Which minute to query? 6
Data at minute 6 is a 7-minute high
Which minute to query? 4
Data at minute 4 is a 1-minute high
Which minute to query? 8
Data at minute 8 is a 1-minute high
Which minute to query? 9
Data at minute 9 is a 3-minute high
Which minute to query? exit
```

If the filename supplied to your function is incorrect or invalid, the return value of your function should be -1. In all other cases the return value should equal to the actual number of data items read from the file and stored in the array that processes this data.

The sensor data could be also be corrupted. For example, the data may not be within the valid range, or contain non-numerical values. When the user queries a minute whose sensor data is corrupted, the program should report, “`Data at minute X is corrupted.`” Meanwhile, all corrupted data are treated as a minimum value. For example, given another file with corrupted data as shown below:

37.8	a.38	139.1	abc.5	37.9	38.8	40.5	39.0	36.9	39.8
0	1	2	3	4	5	6	7	8	9

Query at minute 1, 2, and 3 will all return “**`Data at minute X is corrupted`**” (where X is 1, 2, or 3). Query at minute 4, returns “**`Data at minute 4 is a 5-minute(s) high`**”.

The program also needs to handle user input errors. If the user queries a minute-index that does not exist in the file, the program should produce the error message, “**`Query minute out of range`**” If user types in a string (other than “exit”) that contains non-numeric values (like 11a) or non-integer values (like 15.2), the program should produce an error message “**`Wrong query input`**”

Notes:

- You may open the data file only once and the file must be closed before the function starts accepting user queries. This means the function should not read again from the file while processing user queries.
- Every input item read from the file should be treated as a stream of characters (string), you can use functions like sscanf() or strtod() to convert a string value into a floating point number (invalid data can be set to a value lower than the lowest minimum to identify it as corrupt). You are also free to do the conversion using any other library function if you prefer (the function atof() is deprecated so you may have to use it at your own risk).
- There are no specified limits for the number of data entries stored in the file, the limiting size of the program is the number that’s specified to the function as an argument.
- Do not use break or continue statements in your code, they are highly discouraged.
- **The target in your makefile to build this the object file required should be called “calSpan.o”. By issuing the following make command you should be able to create just the object file:**
  make calSpan.o
  In order for you to compile and link your own main() function with the above object file, *(let’s say the main() function is contained in a file called calMain.c)*:
  gcc calSpan.o calMain.c -o calMain
  will create an executable called calMain. Alternately you can create a target in your makefile to do this also (not required for submission).

Program 2

Files to be submitted: **dataDump.c, (common) makefile**

There is a file called dataDump.h in the public folder which shows the signature of your function dataDump(). You do not submit this header file.

Write a C function that will display the contents of a file to stdout in a manner similar to the octal dump **od** command-line Linux utility. (To learn more about **od**, in a command-line window type **man od** on a machine running Linux). One difference between your output and the one from **od** is that the first column that displays the count of number of bytes read from the start the file is in decimal in your case (**od** displays it in octal). The actual data should be displayed in hex always one byte at a time, with a total of 16 bytes on each line. The other difference is that your function should also show the printable version of the 16 byes of each line as the last column (printable ASCII characters are between 30 and 126, any non printable characters should be printed as a ".").

The makefile must build an object file called "dataDump.o" by using a target called "dataDump.o".

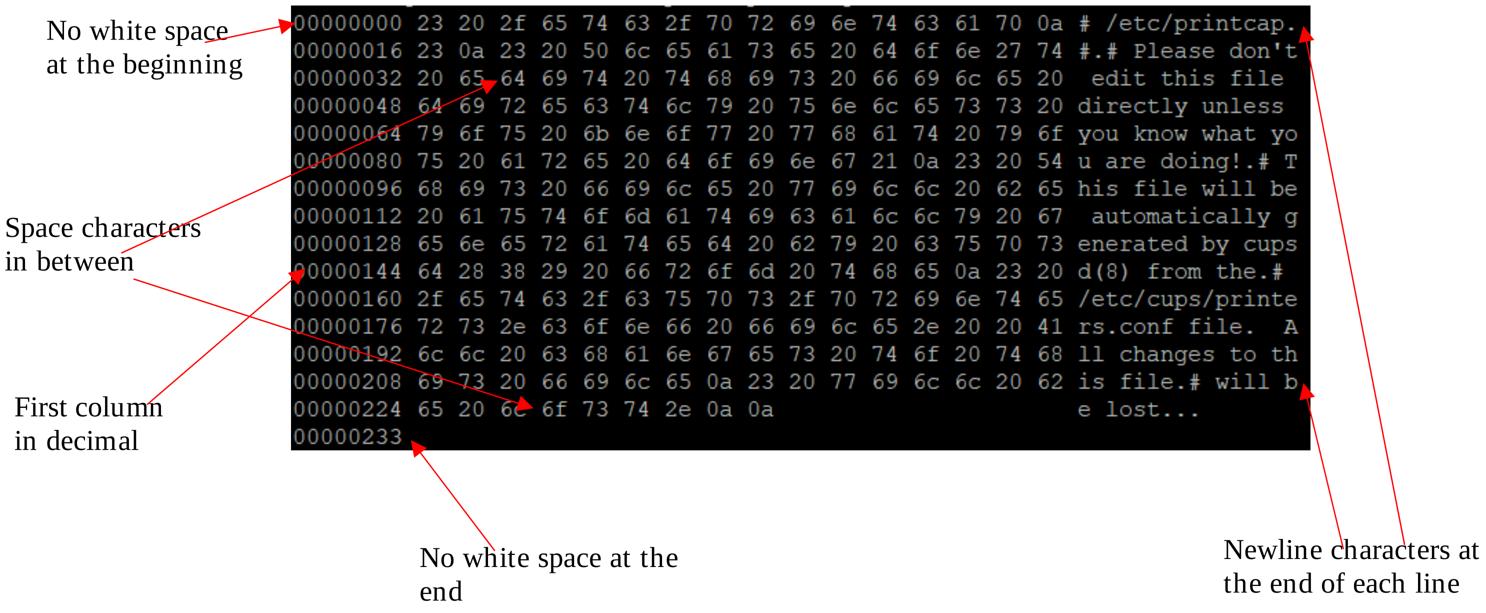
The input file name is passed as an argument to the function. Your function should handle file open errors and return either -1 (if the file was not able to be opened) or the total size in bytes read from the input file.

While you do not submit a file that contains a main() function, you will need to test your implementation by writing your own main() function.

A total of 16 bytes from the input file name must be printed per line. Note that it's easy to see what the octal dump od output of a file displayed one byte at a time in hex:
od -tx1 /etc/printcap

In order to display the printable ASCII characters on the rightmost column you will need to keep appending the bytes to a character array for every line (up to 16 characters) and print them out at the end of the line before you begin the next set of 16 bytes.

There is an input data file called printcap (it's a text file) in the public folder (and the corresponding output that is generated by the function which is also shown below). If your function is called to dump the contents of this file your output should resemble this as closely as possible:


No white space at the beginning
Space characters in between
First column in decimal
No white space at the end
Newline characters at the end of each line

```
00000000 23 20 2f 65 74 63 2f 70 72 69 6e 74 63 61 70 0a # /etc/printcap.
00000016 23 0a 23 20 50 6c 65 61 73 65 20 64 6f 6e 27 74 #.# Please don't
00000032 20 65 64 69 74 20 74 68 69 73 20 66 69 6c 65 20  edit this file
00000048 64 69 72 65 63 74 6c 79 20 75 6e 6c 65 73 73 20 directly unless
00000064 79 6f 75 20 6b 6e 6f 77 20 77 68 61 74 20 79 6f you know what yo
00000080 75 20 61 72 65 20 64 6f 69 6e 67 21 0a 23 20 54 u are doing!.# T
00000096 68 69 73 20 66 69 6c 65 20 77 69 6c 6c 20 62 65 his file will be
00000112 20 61 75 74 6f 6d 61 74 69 63 61 6c 6c 79 20 67  automatically g
00000128 65 6e 65 72 61 74 65 64 20 62 79 20 63 75 70 73 enerated by cups
00000144 64 28 38 29 20 66 72 6f 6d 20 74 68 65 0a 23 20 d(8) from the.#
00000160 2f 65 74 63 2f 63 75 70 73 2f 70 72 69 6e 74 65 /etc/cups/printe
00000176 72 73 2e 63 6f 6e 66 20 66 69 6c 65 2e 20 20 41 rs.conf file.  A
00000192 6c 6c 20 63 68 61 6e 67 65 73 20 74 6f 20 74 68 ll changes to th
00000208 69 73 20 66 69 6c 65 0a 23 20 77 69 6c 6c 20 62 is file.# will b
00000224 65 20 6c 6f 73 74 2e 0a 0a                      e lost...
00000233
```

Notes:

- Although there are several ways to read information from a file, I recommend that you use the library function getc() (or fgetc()) which gets the next byte from a specified stream (file pointer in this case) and advances the pointer to the next byte.
- The public folder contains two input files: printable (this is a text file), and utf16-be (this is a binary file). Run your program against these two files on agate and compare your output versus the expected output for these two files, also in the public folder. These two files will be used as your first two test cases on your program submission.
- **The target in your makefile to build this the object file required should be called "dataDump.o". By issuing the following make command you should be able to create just the object file:**
  make dataDump.o