**CPSC 1070 Lab 3**

The purpose of this Lab is to give you additional practice in working with external files.

In this lab, you will:

* Create a data file
* Practice redirecting standard input and standard output
* Practice reading from and writing to external files
* Process input files when the number of input values is not known
* Get the filename from the command-line

**Background**

A program's working memory of stack and heap data structures only exists while the program is running. To store data between runs, and to capture output, we can use files. A file is like a named array of bytes, and once created a file will persist until deleted, even when the computer is turned off.

Files are identified by a path , which is a generalization of a filename and can be any of the following:

1. a simple filename, e.g. "myfile.jpg";
2. a relative path that specifies a location in the filesystem's directory structure relative to the current working directory, e.g. "mylabs/lab3/lab3.dat"
3. an absolute path that begins with a '/' and fully specifies a location in the directory structure, e.g. "/home/rlowe/mylabs/lab3/lab3.dat"

Each file in a program has both an internal name and an external name. The internal name is what is used in the program; the external name is the name that the operating system knows. These two names must be associated with one another. This association is called *binding* and is done in the function ***fopen***. After processing the file, close the file using the function ***fclose***.

Functions for reading and writing files exist in *stdio.h*. Currently, you have a preprocessor directive in your source file to include ***stdio.h***. The table below contains basic functions needed for opening a file, reading from a file, writing to a file, and closing a file.

|  |  |
| --- | --- |
| How to declare and open a file for output | FILE \*outfile;  outfile = fopen("myoutput.txt", "w");     OR  FILE \*outfile = fopen("myoutput.txt", "w"); |
| How to declare and open a file for input | FILE \*infile;  infile = fopen("myinput.txt", "r");     OR  FILE \*infile = fopen("myinput.txt", "r"); |
| How to read from an input file | fscanf(file\_pointer, fmt-string, addr-list);      e.g. fscanf(infile, "%d %d", &x, &y); |
| How to write to an output file | fprintf(file\_pointer, fmt-string, value-list);      e.g. fprintf(outfile, "%d + %d = %d\n", x, y, sum); |
| How to close a file | fclose(outfile);  fclose(infile); |

**Task 1 - Creating a Data File**

Use your favorite text editor to create a data file named *lab3.dat* containing integer data similar to the following:

15

-30

47

11

-13

19

0

28

16

-14

21

6

10

0

45

17

-32

57

83

-66

0

-71

34

49

72

-40

-38

-19

55

0

23

8

The important thing is to make sure the file contains several positive integers, several negative integers, and a few zero values.

**Redirecting standard input and standard output (Review)**

One way to get input into a program or to display output from a program is to use standard input and standard output, respectively. That simply means to read in data, we use *scanf()* (or a few other functions) and to write out data, we use *printf()*.

If your input is coming from the keyboard, someone is sitting at the keyboard waiting to enter the values at the proper time. This can be tiresome and error prone if the user has to enter a lot of data. Instead of having the user type data at the keyboard, we can take input from a file by using input redirection:

./a.out < inputfile

This allows us to use the same *scanf()* calls we use to read from the keyboard. With input redirection, the operating system causes input to come from the file (e.g., inputfile above) instead of the keyboard.

Similarly, there is output redirection:

./a.out > outputfile

that allows us to use *printf()* as before, but that causes the output of the program to go to a file (e.g., outputfile above) instead of the screen.

Of course, the 2 types of redirection can be used at the same time...

./a.out < inputfile > outputfile

**Exercise 1**

1. Put the following program in a file named *main.c*:
2. #include <stdio.h>
3. int main()
4. {
5. int i;
6. int x;
8. for(i = 0; i < 10; i++)
9. {
10. scanf("%d", &x);
11. printf("%d\n", x);
12. }
13. return 0;

}

1. Compile the program
2. Execute the program, using input and output redirection to read the first 10 values from the *lab3.dat* data file that you created in Task 1, and write those values to the output file *out.txt*
3. Examine *out.txt* to make sure that everything worked correctly.

**External File I/O and Command-line arguments**

While redirection is very useful, and works fine if we have only one input file and only one output file, it is really part of the operating system (not C). In fact, C has a general mechanism for reading and writing files, which is more flexible than redirection alone.

For C File I/O you need to use a FILE pointer, e.g. *FILE \*fpIn*, which will let the program keep track of the file being accessed. (You can think of it as the memory address of the file or the location of the file).

**Processing data when we do not know how many values are in the file**

Sometimes we know exactly how many data values are to be processed; however, there will be times when we will not know the number of elements to be processed. There are basically three techniques that we can use when the number of data values is not known: sentinel-controlled loops, end-of-data controlled loops, and end-of-file controlled loops. For each technique, we want to do the following:

1. read before entering the loop
2. test the loop condition
3. process the value(s) read
4. read before entering the loop again.

It is important to note that end-of-file is not set until there is an attempt to read and the file does not contain any more values. Similarly, the return value of fscanf() does not return an incorrect value until there is an attempt to read and the file does not contain the correct number of expected values. That is why it is important to read **before** entering the loop; otherwise, there must be a check within the loop to make sure we have not reached the end of the file. If there is no read before entering the loop and no check within the loop, then the program will attempt to process one too many values.

An end-of-data controlled loop tests against the number of values we expect to be read by fscanf() each time. To implement an end-of-data loop, use the value returned by *fscanf* to control the loop. e.g.

int n;

n = fscanf(infile, "%d", &val); // priming read

while(n == expected\_number\_of\_input\_values) // test condition

{

// process

n = fscanf(infile, "%d", &val); // read again

}

alternatively, you can use

while(fscanf(infile, "%d", &val) == expected\_number) // read and test

{ // before entering loop

// process

}

Another way to test for end of file is with the *feof()* library function. The function takes a file pointer and returns 1 if the previous read operation reached the end of the file.

An eof-contolled loop works similar to an end-of-data loop: for an eof-controlled loop, modify the condition of the while to be:*while(!feof(infile))*

**Task 2 - Lab 3 Assignment**

For this task, assume you do not know how many data values are in *lab3.dat*.

Use your favorite editor to create a file named *lab3.c*. You are to write a C program that reads all values in the input file. While reading the file, the program should count the number of positive integers, the number of negative integers, and the number of zeros, and also write the positive numbers to the positives output file and the negative numbers to the negatives output file. These file names will be given as command-line arguments. ***Do not*** write the zeros to a file.

After consuming all input data, the program should print the positive count, with label, to the positives output file, print the negative count, with label, to the negatives output file, and print the number of zeros to *stdout*, then close all files. Do not label any other values in the output files.

The input file name, the output file name for positive values, and the output file name for negative values are command-line arguments, in that order. For the input file, use the *lab3.dat* data file that you created in Task 1.

Since you will be using command-line arguments, use

*int main(int argc, char \* argv[])*

argv[1] will point to the input file name   
argv[2] will point to the positives output file name, etc.

You will need to do the following:

1. Declare FILE pointers for the input file, output of positives file, and output of negatives file.
2. Verify sufficient command-line arguments
3. Open each file and verify that the open was successful.
4. Process
5. Close all files

**Testing your solution**

1. Compile lab3.c

gcc -Wall -o lab3 lab3.c

1. Execute the program using:

*./lab3 lab3.dat positives.txt negatives.txt*

1. Check the output files to make sure they are correct.

**Submission**

Use [Handin](https://handin.cs.clemson.edu/) to submit lab3.c and lab3.dat

It is your responsibility to make sure that your files are submitted to the appropriate handin folder. Check your handin folder after submitting your files.