CSCE 155 - C

Lab 10.0 - File I/O

Prior to Lab

Before attending this lab:

1. Read and familiarize yourself with this handout.
2. Read Chapters 9 and 22 of the Computer Science I textbook
3. Watch Videos 10.1 thru 10.4 of the Computer Science I video series

Peer Programming Pair-Up

For students in the online section: you may complete the lab on your own if you wish or you may team up with a partner of your choosing, or, you may consult with a lab instructor to get teamed up online (via Zoom).

For students in the face-to-face section: your lab instructor will team you up with a partner.

To encourage collaboration and a team environment, labs are be structured in a *peer programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Function	Purpose
`fopen()`	Opens a file for reading (r) or writing (w)
`fclose()`	Closes the file stream passed to the function
`fprintf` , `fscanf`	Output/input from a file stream in a structured manner
`fgets()` , `fputs()`	Inputs/outputs an entire string (line) from/to a file
`fwrite()` , `fread()`	Inputs/outputs binary data

Table 1: Several File I/O Functions

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

1 Lab Objectives & Topics

At the end of this lab you should be familiar with the following

- Understand the differences between binary and plaintext data
- How to read from a file and process information
- How to write to a file to persist information
- Have some exposure to other topics such as XML and sorting

2 Background

The life span of most program is short–measured in seconds or microseconds. For data to be useful, it needs to last beyond the typical program. This is known as data persistence. One mechanism for persisting data is to store it in a file. Files can generally consist of raw binary data or plaintext. In either case, the data needs to be structured in some manner for a program to be able to read and write it.

In C, file I/O (input/output) is facilitated through several standard functions provided by the standard input/output (`<stdio.h>`) library, including those found in Table 1.

You will use these functions to complete the activities in this lab. For more details, refer to a standard C reference on how to use these functions.

String Tokenizing

Often times it is necessary to process formatted data. One common solution is to tokenize a string along some delimiter. For example, given a string of comma-separated-values (CSV) such as `Sandberg,Ryne,1060 West Addison Street,Chicago,IL,60613` we may want to process each field, so we would want to split the string using the comma as a delimiter into its constituent components (last name, first name, address, city, state, zip for example).

The standard string library provides a function for doing this:

```
char *strtok(char *str, const char *delim);
```

The way to use it is as follows: the first argument is the string to be tokenized, the second is the delimiter (or list of delimiters to tokenize on). The return value is the next token with the delimiter removed. The first time you call `strtok`, the first argument should be the string to be tokenized. Subsequent calls should use pass `NULL` to tokenize on the same string (otherwise tokenization begins anew on the string). The same delimiter should be passed each time. The tokenization ends when `strtok` returns `NULL`. Note: the observant coder will notice that the first argument is *not* a `const char *`. As such you can expect `strtok` to *change* the provided string as it tokenizes, so be careful.

3 Activities

Clone the code for this lab from GitHub using the following URL: https://github.com/cbourke/CSCE155-C-Lab10.

3.1 Plaintext versus Binary Data

In general, data files can contain either binary data (a collection of 0s and 1s) or plaintext data (ASCII). Binary data is generally readable only by a computer or program that interprets the 0s and 1s as different types of data (7-bit ASCII characters, 32-bit integers, etc.) while ASCII text is readable by humans, but may need additional formatting and data conversions to be handled by a program. In this first activity you will get some experience in the contrast between these two types of data.

Instructions

In this exercise, you will work with a pre-written program that opens a file containing census data on states from the 2010 census.

1. Open the `stateData.txt` data file which is in the `data` folder and observe its contents (do not edit this file)

2. Examine what the `stateData.c` program is doing; compile and run it
3. The program will create a binary output file, `stateData.dat` in the same `data` directory.
   a) The unix command `file` is a utility that determines the type of a file; examples:

   ```
   file stateData.txt
   file stateData.dat
   ```

   b) Open the `stateData.dat` file in your preferred text editor and observe its contents
   c) The unix command `ls -la` will list some detailed statistics of all the files in a directory. Run it and observe the sizes of the files (5th column in bytes).
4. Answer the questions on your worksheet and move on to the next activity.

3.2 File Output

Extensible Markup Language (XML) is a markup language that defines a set of rules for formatting data in a file that is both human readable and can be processed by a machine. Each piece of data is semantically marked-up to indicate what that data represents. This enables data to be more portable and interoperable across different programs and different programming languages. Many tools and frameworks have been developed around its usage. For example, the state population data may be encoded in XML as follows.

```
<STATES>
  <STATE>
    <NAME>Nebraska</NAME>
    <POPULATION>1826341</POPULATION>
  </STATE>
  ...
  <STATE>
    <NAME>Ohio</NAME>
    <POPULATION>11536504</POPULATION>
  </STATE>
</STATES>
```

Instructions

Modify the `stateData.c` program by implementing (and calling) the function:

```
void toXMLFile(char states[][35], int *populations, int numStates);
```

1. The function should open a file for writing, `stateData.xml`
2. It should create an XML file containing marked up data on all 50 states as in the example above.
3. Hint: a convenience function, `rtrim` has been provided to you that trims the trailing whitespace from a string and returns a pointer to the new, trimmed string.
4. Answer the questions on your worksheet and demonstrate your program to a lab instructor.

3.3 File Input & Data Processing

Reading data from a file is often done in order to process and aggregate it to get additional results. In this activity you will read in data from a file containing win/loss data from the 2011 Major League Baseball season. Specifically, the file `data/mlb_nl_2011.txt` contains data about each National League team. Each line contains a team name followed by the number of wins and number of losses during the 2011 season. You will open this file and process the information to output a list of teams followed by their win percentage (number of wins divided by the total number of games) from highest to lowest.

Instructions

1. Open the `mlb.c` C source files. Much of the program has already been provided for you, including a convenience function to sort the lists of teams and their win percentages as well as a function to output them.
2. Add code to open the data file and read in the team names, wins and losses and populate the `teams[]` and `winPercentages[]` arrays with the appropriate data
3. Call the sort and output functions to sort and display your results
4. Answer the questions on your worksheet and demonstrate your working program to a lab instructor

4 Handin/Grader Instructions

1. Hand in your completed files:
   - `stateData.c`
   - `mlb.c`

- `worksheet.md`

through the webhandin (https://cse-apps.unl.edu/handin) using your cse login and password.

2. Even if you worked with a partner, you *both* should turn in all files.
3. Verify your program by grading yourself through the webgrader (https://cse.unl.edu/~cse155e/grade/) using the same credentials.
4. Recall that both expected output and your program's output will be displayed. The formatting may differ slightly which is fine. As long as your program successfully compiles, runs and outputs the *same values*, it is considered correct.

5 Advanced Activity (Optional)

1. When we sorted the baseball teams and their win percentages, we had to do all the "bookkeeping" ourselves: that is, we had to swap elements in both arrays to make sure that the i-th team name matched up with the i-th win percentage. A much better way would have been to define a user defined struct type to hold the team name and win percentage in the same structure. Redesign the program to use such a struct.
2. In general, there are no restrictions on the length of a line in a plaintext file (and in a data file, there is not even a concept of a "line"). For the best readability, however, it is best to keep lines to a limited, consistent length. One common maximum length for monotype font is 72 characters per line (CPL). A plaintext file, `star_wars.txt` has been provided (the original draft script of the movie Star Wars) that contains some very long lines. Write a program to read in the file, line by line. If the line exceeds the 72 CPL limit, break it up into multiple lines (but do not break up individual words). Output the resulting well-formatted file to a separate file.