The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 214– Programming Principles and Practice

Assignment 5

Date Due: October 26, 2020

Total Marks: 32

Submission Instructions

- Assignments must be submitted using Canvas.
- Programs must be written in C conforming to the C11 standard.
- Always include the following identification in your solutions: your name, NSID, student ID, instructor's name, course name, and course section number.
- No late assignments will be accepted. See the course syllabus for the full late assignment policy for this class.
- **VERY IMPORTANT**: Canvas is very fragile when it comes to submitting multiple files. We insist that you package all of the files for all questions for the entire assignment into a **ZIP** archive file. This can be done with a feature built into the Windows explorer (Windows), or with the **zip terminal command** (LINUX and Mac). We **cannot accept** any other archive formats. This means no tar, no gzip, no 7zip. Non-zip archives will not be graded. We will not grade assignments if these submission instructions are not followed.
- Instructions on "how to create zip archives" can be found here `https://canvas.usask.ca/courses/9771/pages/how-to-zip-slash-compress-your-files`

1 Background

In this background section, we are going to learn about the `feof()` function of library header `stdio.h`, and how to pass command line arguments to C programs.

1.1 `feof()` function in C

1.1.1 Usage:

The feof function is used to determine whether the end of a file has been reached. Its prototype is as follows:

```
int feof(FILE* fileptr);
```

The argument to this function, "fileptr", is a pointer to an open file, such as returned by fopen. The function returns an integer: 1 (true) if the end of the file (eof) has been reached, and 0 (false) otherwise.

1.1.2 Details:

The `feof()` function returns 1 **only if the most recent operation on the file** `fileptr` **encountered the end of the file**. For example, if the most recent file operation was `fscanf()`, and it was not able to read a value specified because the end of the file was reached first, then a subsequent call to `feof()` would return 1.

It's important to note that `feof()` doesn't actually read the file referenced by `fileptr`. It only checks whether the **previous** operation reached the end of the file.

Example 1

Here is an example showing the use of feof:

```
#include<stdio.h>

int main()
{
    FILE *infile = NULL;
    char buf[50];
    infile = fopen("sales.txt","r");

    if(infile == NULL)
    {
      printf("Error␣opening␣file.\n");
      return 0;
    }

   // Read the first line
   fgets(buf, sizeof(buf), infile);

   // If that didn't reach the end of file, keep reading ...
    while(!feof(infile))
   {
      // Print the read line.
      printf("%s\n", buf);

      // Read the next line
```

```
        fgets(buf, sizeof(buf), infile);
    }
    fclose(infile);

    return 0;
}
```

You might be wondering if both `fgets()` calls are really needed. Indeed, it is critical that they are both there because of the fact that `feof()` must be checked **after** the operation that might have reached end of file, but **before** any data that might have resulted from the potentially failed I/O operation is used. The example above performs an initial file read, then checks whether that succeeded, and if it did, prints the data, then gets another line, then returns to the top of the loop to immediately check if that read was successful.

Many beginners want to omit the initial `fgets()` and do this:

```
    while(!feof(infile))
    {
        // Read the next line
        fgets(buf, sizeof(buf), infile);

        // Print the read line.
        printf("%s\n", buf);
    }
```

But this is **incorrect**. Firstly, `feof()` is called before any operations on the file are performed. Secondly, the `printf()` occurs immediately after the `fgets()` so the line is printed **before** we have checked whether the end of file was reached by the `fgets()`. Only after the `printf()` do we check whether `fgets()` failed when we return to the top of the loop. The same principle applies when using `fscanf()`. The test for `feof()` must come **after** the use of `fscanf()` but **before** any of the data read by the attempted `fscanf()` is used (because if the `fscanf()` failed due to end of file, the data will be bad!).

1.1.3 The `rewind()` function.

The `rewind()` function defined in `stdio.h` has the following prototype:

```
void rewind(FILE *stream);
```

This function resets the open file called `stream` that has been opened for reading so that subsequent reads will start from the beginning of the file again.

1.2 Command Line Arguments to Programs

We know that you can pass arguments to terminal commands. Indeed, when we use

```
    gcc -o myprog -Wall -Wextra main.c
```

everything after `gcc` is a *command line argument*. In this section we will learn how to access the command line arguments passed to a C program.

Command line arguments are passed as arguments to the `main()` function. When we want to access such argments, we have to declare `main()` to have two parameters, an integer which indicates the number of command line arguments called `argc`, and a pointer array of length `argc` where each element is a pointer to an array of characters containing a string. Each string is one of the command line arguments. Thus, to receive command line arguments, main must be declared like this:

```
int main(int argc, char *argv[]) {
    ...
}
```

Note that, when writing your programs, you will either use a main() function with no arguments or a main() function as described above. If your main() function has no arguments, any command line arguments that are passed in will simply be ignored by your program. These are the only two valid ways to declare main().

Once execution starts in `main()`, `argc` is the number of command line arguments, and `argc[i]` is the *i*-th command line argument as a string. The type of `argc[i]` is, of course, `char*`.

The string `argv[0]` is a bit special, it's not actually one of the command's arguments, it's the name of the command itself. If the command given was:

```
myprogram inputfile.txt outputfile.txt
```

then `argv[0]` would be the string `"myprogram"`, `argv[1]` would be the string `"inputfile.txt"`, and `argv[2]` would be the string `"outputfile.txt"`.

Example 2

```
#include <stdio.h>

int main(int argc, char *argv[])
{
     int i = 0;
     printf("cmdline␣args␣count=%d\n", argc);

    /* First argument is always the executable name */
    printf("exe␣name=%s\n", argv[0]);
    for (i=1; i< argc; i++)
    {
        printf("arg%d=%s\n", i, argv[i]);
    }
    printf("\n");
    return 0;
}
```

When the program is executed, the output is,

```
./test first second third

cmdline args count=4
exe name=./test
arg1=first
arg2=second
arg3=third
```

In above output, the total argument count stored in `argc` is 4 because the command line contained the program name and 3 argument for a total of 4. The parameter `argv` of `main()`, holds the name of the executable program and the last three are arguments (first, second, third) passed to the program.

1.2.1 Numeric Command Line Arguments

Command line arguments are always stored in `argv` as strings. But numeric arguments can be converted from their string representations to numeric types using library functions that are defined in `stdlib.h`.

The function `atoi()` takes a string as an argument, and returns (if possible) its the integer value represented by the string. The prototype is:

```
int atoi(const char* str);
```

Similarly, the function `atof()` takes a string as an argument, and returns (if possible) the type `double` represented by the string. The prototype is:

```
double atof(const char* str);
```

The next example obtains all command line arguments as character strings and converts them to integers and floats.

Example 3

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
   int i = 0;
   int d;
   double f;
   long int l;
   FILE *infile = NULL;
   printf("\ncmdline␣args␣count␣=␣%d", argc);

   /* Executable name is the first argument */
   printf("\nexe␣name␣=␣%s", argv[0]);

   // Print the remaining arguments.
   for (i=1; i< argc; i++)
   {
      printf("\narg%d=%s", i, argv[i]);
   }

   /* Convert the first argument from string to an integer. */
   d = atoi(argv[1]);
   printf("\nargv[1]␣as␣integer␣=␣%d",d);

   /* Convert the first argument from string to a double. */
   f = atof(argv[2]);
   printf("\nargv[2]␣as␣float␣=␣%f",f);

   printf("\n");
   return 0;
}
```

The output of this example program is,

```
   ./test 456 150

   cmdline args count = 3
   exe name = ./test
   arg1=456
   arg2=150
   argv[1] as integer = 456
   argv[2] as float = 150.000000
```

2 Assignment Problems

Alert: Take note that on this assignment we are requiring code commenting and have assigned points to commenting in the grading rubrics.

Question 1 (8 points):

Purpose: To practice accessing of arrays and command line arguments.

Sam started a "supermarket on wheels" to help his neighborhood in this COVID-19 pandemic. Sam soon realized that people prefer buying from his 'moving store' rather than visiting to the supermarket and he started maintaining a product performance report for each category. Ideally, this report should tell him the summary of items sold on weekdays per category. Then Sam use the product performance report to determine which items are worth investing in and which ones shouldn't be re-ordered. Revisiting the report for last week, he realized that he made a mistake in entering items in dairy and fruit category. Watermelon has been entered in dairy report under the Yogurt column, and and yogurt sales have been entered in fruit report under the Watermelon column. Sam is trying to rearrange the selling history of yogurt and watermelon.

Input File Format

The format of the data files `fruit.txt` and `dairy.txt` is as follows.

- The first line is a column header row containing column titles. Each column title is a string of less than 100 characters containing no spaces referring to a type of fruit (in `fruit.txt`) or a dairy product (in `dairy.txt`). In this case you can assume that the two input files have the same number of lines.
- Each subsequent line contains six data items per line separated by spaces. In order, they are:
  - The day of the week as a string containing no spaces with a length of less than 25 characters.
  - Five additional items in each line that are the numbers of each food item (column title refers to the food item) is sold in as a string with length less than 15 characters.
- The number of columns in the file is fixed at 6, but the number of lines in the file is arbitrary and could be anything.

Your Tasks

The goal of your program is to swap the values in the `Watermelon` column in the dairy report with the values in the `Yogurt` column in the fruit report.

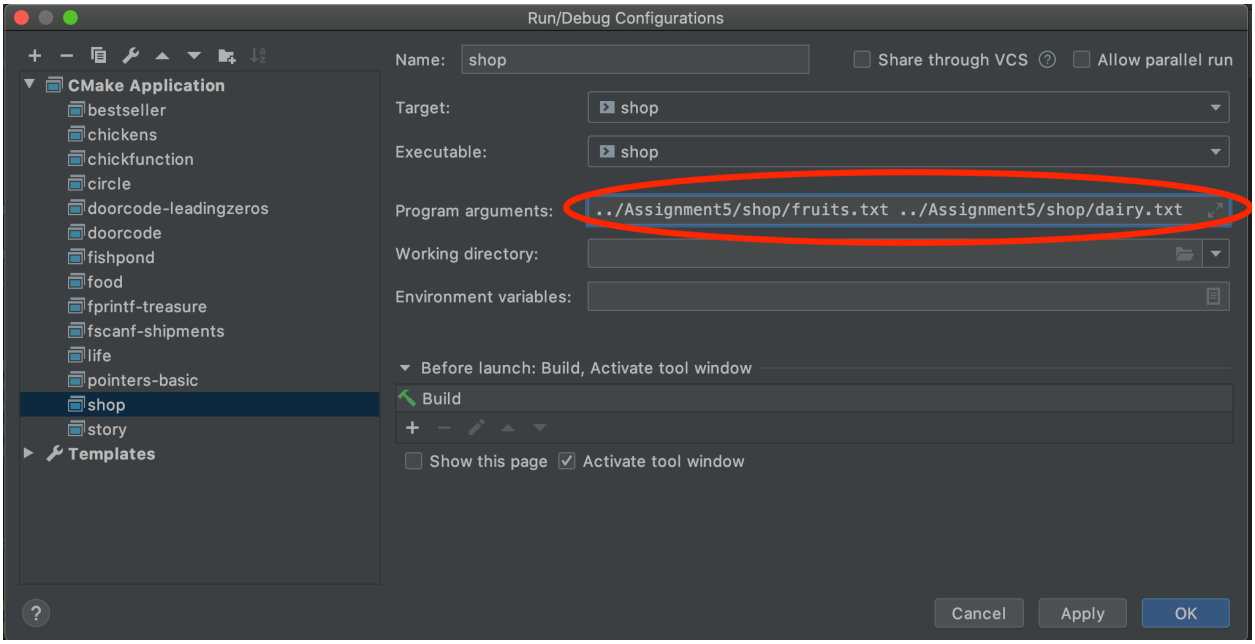
(a) Write a `main()` function with parameters `argc` and `argv` that accept the names of two files (fruit.txt, dairy.txt, with or without file path) as arguments. Make sure you check for the condition that at least two arguments were provided (one for each filename) and if not, take appropriate action.

(b) Read the two input data files, swap the data in the Watermelon column of `fruits.txt` with the Yogurt column of `dairy.txt` and write the corrected data to `fruits-revised.txt` and `dairy-revised.txt` respectively. You may assume that Watermelon is the last column of `fruits.txt` and that the Yogurt column is the last column of `dairy.txt`. Take care that the header row of each file is not modified.
Hints: There are several different approaches to this with varying degrees of difficulty and complexity. Think before you code. Think especially about what data needs to be stored, and how much of the data really needs to be stored simultaneously.

(c) Use the `feof()` function while reading lines to detect the end of the file since the number of rows in the files are unknown and may vary.

Hints

You will not be able to run a program that requires command line arguments from CLion without special configuration. The easiest thing to do for this question is to both compile and run your program from the command line. If you really insist on using CLion, the easiest thing to do is to compile with CLion, then open a terminal and go into the cmake-build-debug folder in your CLion project and run the compiled executable file there directly from the command line. To make it run directly from CLion you need to find the drop-down beside the "run" button (green triangle) at the top-right of the window and edit the run configuration of your program. Click on the drop-down, choose "Edit configurations", choose the name of your program from the list on the left, then fill in the command line arguments where it say "Program Arguments" in the space on the right. If it is not obvious how to do this, please run your program from the command line.



Question 2 (12 points):

Purpose: To practice 2d arrays, and 1D views of 2D arrays.

Using the revised files from question 1, help Sam to figure out which items are worth investing by sorting the selling history of each item. So that Sam can use this details while ordering items.

Input File Format

The files using for this question are the output files `fruit-revised.txt` and `dairy-revised.txt` from question 1. The format of the data files is same as question 1.

You can use your own output files from question 1, or, if you want to attempt this question without having finished question 1, you can use the question 1 output files provided.

Your Tasks

(a) Write `main()` function with `argc` and `argv` parameters that accept the name of files (fruit-revised.txt and dairy-revised.txt with or without file path) as arguments. Make sure you check for the condition that at least two arguments were provided (one for each filename) and if not, take appropriate action.

(b) Write a function that counts how many lines of text there are in a file. Your function should take pointer to a newly opened file (i.e. a pointer to type `FILE` as a parameter, read the file a line at a time, and determine return an integer indicating how many lines are in it.

(c) In `main()` use the function in part (b) to determine how many lines are in each file and make sure that the number of lines is at least 2, and that the number of lines in the two files are the same. If not, terminate the program. Then use `rewind()` (described in the background section, above) to reset the open files so they can be read again (do not close them first).

(d) Prepare your data structures to hold the data in the files. For each file, create a two dimensional array of characters to store the column headers of each file. You need to store column headers for 6 columns. For each file, dynamically allocate a one-dimensional view of a two dimensional array that can hold all of the unsigned integer data in the file (i.e. do not include the first row and the first column, allocate only space for the numeric data in the files). The dynamically allocated arrays will need 5 columns (one for each product) and a number of rows equal to one less than the number of rows in the files.

(e) Read the column headers of each file into your 2D arrays of strings. Read the numeric data from the files into your 1D views of 2D arrays.
Hint: Remember, you'll have to use linear indexing for your 1D views of 2D arrays.

(f) Write a function called `display_sales()` which takes three parameters: the column headers in a file as a 2D array of characters, the numeric data from a file (as a 1D view of a 2D array) and an unsigned integer indicating the number of rows in the 1D view of the 2D array (the number of columns is fixed at 5). The function should compute the total sales of each product in each column, and then print to the console the name of the product and the total sales (see sample output, below). In other words, you are computing the sum of each column of numeric data and reporting this sum along with the corresponding product name. This function should return nothing.

(g) From `main()`, call the `display_sales()` function twice, once for the data in each input file, to display the total sales of each product represented in the file (see sample output).

Sample Output

```
Total Banana sales: 197
Total Apple sales: 158
Total Orange sales: 192
Total Mango sales: 140
Total Watermelon sales: 918
Total Milk sales: 247
Total Cheese sales: 168
Total Butter sales: 212
Total Cream sales: 150
Total Yogurt sales: 177
```

Question 3 (12 points):

Purpose: To practice structures and linked data structures with pointers.

You are provided with a file named code.txt that contains three letter train station codes in each line. Create a linked list (i.e. a node chain) to add each of the train stations as a node in the same order in which it is provided. Once the list is built, develop an additional function which allows the user to search for a user-specified three letter code and then output the next two upcoming station codes to the one entered by the user. We assume that you have seen Node Chains and Linked Lists from CMPT 145.

Your program will use the following declarations:

```
#define STATION_CODE_LENGTH 4

typedef struct _station {
    char code[STATION_CODE_LENGTH];
    struct _station *next;
} Station;


typedef struct _route {
    Station *first_station;
    unsigned int num_stations;
} Route;
```

The `Route` structure holds a pointer to the first station in the linked list, and the total number of stations in the list.

The `Station` structure holds the station's three-letter code, and a pointer to the next station in the linked list.

Your Tasks

(a) Write a function called `read_stations()` that takes a parameter of the `FILE*` which is a pointer to a freshly opened file that contains one three-letter station code per line. Read the codes in the file and return a pointer to a new dynamically allocated `Route` structure that contains a linked list of `Station` nodes in the reverse order that they appear in the file. Each from the file code is stored in a `Station` structure.
Hint: as you read each station code, dynamically allocate a new `Station` *structure for it, make the existing first station on the route the new station's successor, and then set the first station in the route to be the new first station. In effect, you'll be adding each new station to the front of the existing linked list, which reverses the order of the codes from how they appear in the file. Remember: Each* `Station` *structure holds a pointer to the next* `Station` *structure in the linked list, and the* `Route` *structure, of which there is only one, holds a pointer to the first* `Station` *in the list.*

(b) Write a function `display()` that takes a pointer to a `Route` structure, and prints to the console every station code in the list.
Hint: Follow the chain of pointers to `Station` *structures and print out each code.*

(c) Write another function `search()` that takes a pointer to a `Route` structure and a 3-charater station code *c* as a string as arguments and displays the position number of the station whose code matches *c*, and the next two upcoming stops (if they exist). See sample output, below. If *c* does not exist in the list, print an appropriate message. The search is case sensitive. For example, `yyz` does not match `YYZ`.

(d) In main(),

- Open the input file `code.txt` for reading.
- Call `read_stations()` from part (a) to create a `Route` linked list structure.
- Call `display()` on the route structure returned by `read_stations()`.
- Prompt the user to enter a train code.
- Call the `search()` function on the `route` returned by `read_stations()` to search for the train code entered by the user.

Sample Output

```
YWG
YYJ
YVR
YYZ
YYT
YXE
YQR
YQB
YOW
YUL
YXU
YHM
YHZ
YQM
YQX
YEG
YYC
Code position: 4
Next two codes:
YYT
YXE

// For the remaining examples, we have omitted the
// list of codes being printed because they will
// always be the same.

Enter code to search: YEG
Code position: 16
Next two codes:
YYC
Beyond end of line.

Enter code to search: YYC
Code position: 17
Next two codes:
Beyond end of line.
Beyond end of line.

Enter code to search: yyz
Code not found in list.
```

3 What to Hand In

Hand in a `.zip` file archive which contains the following files:

asn5q1.c: Your completed solution for question 1.

asn5q2.c: Your completed solution for question 2.

asn5q3.c: Your completed solution for question 3.

VERY IMPORTANT: You **must** hand in a ZIP archive containing all of the above files. You may not use any other type of archive (this means no gzip, no 7zip, etc.), and you may not submit the files individually. We regret this necessary inconvenience but failure to follow these instructions may result in your assignment not being graded. We simply do not have the resources to handle special cases when we have so many students in the class. Instructions on "how to create zip archives" ca be found here `https://canvas.usask.ca/courses/9771/pages/how-to-zip-slash-compress-your-files`.

We will not grade assignments if these submission instructions are not followed.

Grading Rubric

The grading rubric can be found on Canvas.