Lab Week 8: Airports!

In this lab we will practice and learn how to use the following:

- Opening files.
- Command line arguments.
- Reading from files.
- Parsing CSV data files.
- Using structures

The application we will ultimately build is a query system that, given a three-letter airport code, returns its full name and location. Here is an example application:

```
Enter airport code: PHL
PHL – Philadelphia Intl
Philadelphia, PA (USA)

Enter airport code: PHX
PHX – Phoenix Sky Harbor International
Phoenix, AZ (USA)

Enter airport code: LHR
LHR is not a known airport in USA.
```

Our application program will input a database of all the known airports. For each airport, it has the following:

Code: a three letter airport code (e.g. PHL, PHX, BOS, etc.). Also called IATA code.
Name: name of the airport (e.g., Phoenix Sky Harbor International)
City: city where located (e.g., Philadelphia, Phoenix, Tuscaloosa, etc.)
State: state (e.g., PA, AZ, AL, etc.)
Country: USA (we will only use US airports, but some foreign US territories are included!)

To begin, we need to learn how to use several new features of the C language (as discussed in today's class). We will start by writing small programs to try out these features until we are ready to build the complete application.

Task#1: Opening Files in C

First, you should copy the database of airport codes from:
`~dkumar/cs223/Lab8/code.txt`

If you take a peek in the file, you will notice that it contains entries as shown below:

```
#iata,airport,city,state,country
00M,Thigpen ,Bay Springs,MS,USA
00R,Livingston Municipal,Livingston,TX,USA
00V,Meadow Lake,Colorado Springs,CO,USA
01G,Perry-Warsaw,Perry,NY,USA
…
```

The first line in the file (the one that begins with #) describes the format of all the entries that follow. The remainder of the file has one line for each airport containing the details as described above. Can you figure out how many airports are in this file?
(Hint: use the command `wc`).

Our first task then is to learn how to access or open this file through the C program. Carefully read the program below:

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

#define FILE_NAME "code.txt"

int main(void) {

   FILE *fp;

   printf("Opening file: %s\n", FILE_NAME);
   fp = fopen(FILE_NAME, "r");
   if (fp == NULL) {
      printf("Unable to open file %s.\n", FILE_NAME);
      exit(EXIT_FAILURE);
   }
   printf("Able to open file %s\n", FILE_NAME);


   fclose(fp);
   return(0);
} // main()
```

This program attempts to open a file (named, "**code.txt**"). As explained in the class, the names `FILE`, `fopen()` and `fclose()` are available in the **stdio** library. The names `EXIT_FAILURE` (=1) and `exit()` are defined in the **stdlib** library.

Go ahead, enter the code above in a file called **task1.c**
Compile and run the program. Make sure it is able to access and open the file **code.txt**
(It should print the message: `Able to open file code.txt`)

Task#2: Command Line Arguments

In Task#1, we hard-coded the name of the input file (in the name FILE_NAME). Next, we will expect the user to enter the name of the data file on the command line. For example, if our program is called task2 (which it will be, soon!), the user will run it as:

```
./task2 code.txt
```

To begin, make a copy of task1.c into a new file, **task2.c** and modify the program as shown below:

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

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

   FILE *fp;

   if (argc != 2) {
      printf("Cannot proceed: Missing file name.\n");
      exit(EXIT_FAILURE);
   }
   printf("Opening file: %s\n", argv[1]);
   fp = fopen(argv[1], "r");
   if (fp == NULL) {
      printf("Unable to open file %s.\n", argv[1]);
      exit(EXIT_FAILURE);
   }
   printf("Able to open file %s.\n", argv[1]);

   fclose(fp);
   return(0);
} // main()
```

Notice, that we also entered a check to make sure that there was actually a possible file name provided as a command line argument. Remember, the value of `argc` is the total number of arguments received (including the name of the program).

Once ready, run the program several times: with and without command line arguments; with names of data files that are not present, etc.

Task#3: Reading From File

The C **stdio** library command `fgets()` inputs a line as a string as shown below:

```
char line[MAX_LINE_LENGTH];
fgets(line, sizeof(line), fp))
```

The above call reads one line from the file pointed to by `fp` (after it is opened). After the call, line contains all the characters up to and including the end-of-line character (`'\n'`). Additionally, `fgets()` returns a null pointer (`NULL`) when there is no more input (like when an end of file is reached). Also, note that `fgets()` will read only as many as `MAX_LINE_LENGTH - 1` characters, leaving the last spot for a null character (`'\0'`). The program below reads all the lines from a provided input file and then prints out the contents of the last line it read. Go ahead and enter it in a file, **task3.c**

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

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

   FILE *fp;
```

```
   if (argc != 2) {
      printf("Cannot proceed: Missing file name.\n");
      exit(EXIT_FAILURE);
   }
   printf("Opening file: %s\n", argv[1]);
   fp = fopen(argv[1], "r");
   if (fp == NULL) {
      printf("Unable to open file %s.\n", argv[1]);
      exit(EXIT_FAILURE);
   }
   printf("Able to open file %s.\n", argv[1]);
   printf("Reading from file %s.\n", argv[1]);

   char line[80];
   while ( (fgets(line, sizeof(line), fp)) != NULL) {
      int l = strlen(line);
      line[l-1] = '\0'; // Eliminate the EOL char ‘\n’
   }
   printf("%s\n", line);

   fclose(fp);
   return(0);
} // main()
```

Try the program with the input file you have and make sure it prints out the line:

```
ZZV,Zanesville Municipal,Zanesville,OH,USA
```

Task#4: Parsing CSV Data Files

As explained in class, this requires use of the `strtok()` function from the **string** library. For your convenience, we have provided the following functions that you can use:

```
void tokenize(char * str,
      char *code, char * name,
      char *city, char *state, char *country) {

   // Split the string in str into five tokens/strings

   char *token;
   token = strtok(str, ",");
   strcpy(code, token);


   token = strtok(NULL, ",");
   strcpy(name, token);

   token = strtok(NULL, ",");
   strcpy(city, token);

   token = strtok(NULL, ",");
   strcpy(state, token);
```

```
   token = strtok(NULL, ",");
   strcpy(country, token);
} // tokenize()
```

Study the `tokenize()` function carefully and use your class notes to fully understand what it is doing. Then, use it in the function in `main()` as shown below:

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

   char code[30];       // iata code etc.
   char name[40];
   char city[30];
   char state[30];
   char country[30];

   FILE *fp;

   if (argc != 2) {
      printf("Cannot proceed: Missing file name.\n");
      exit(EXIT_FAILURE);
   }
   printf("Opening file: %s\n", argv[1]);
   fp = fopen(argv[1], "r");
   if (fp == NULL) {
      printf("Unable to open file %s.\n", argv[1]);
      exit(EXIT_FAILURE);
   }
   printf("Able to open file %s.\n", argv[1]);
   printf("Reading from file %s.\n", argv[1]);

   char line[80];
   while ( (fgets(line, sizeof(line), fp)) != NULL) {
      int l = strlen(line);
      line[l-1] = '\0';
   }
   printf("%s\n", line);
   printf("Parsing <%s> into tokens...\n", line);
   tokenize(line, code, name, city, state, country);

   printf("%s\n", code);      // Test: Print data from last line
   printf("%s\n", name);
   printf("%s\n", city);
   printf("%s\n", state);
   printf("%s\n", country);

   fclose(fp);
   return(0);
} // main()
```

Again, as before, make a copy of your task3.c into a new version, **task4.c**. Complete the program, compile, run, and test it. You should get the output shown below:

```
ZZV
Zanesville Municipal
Zanesville
OH
USA
```

Task#5: Using Structures

Now, we are ready to define the main data structures for the application. First, we will define an airport:

```
typedef struct {
   char code[5];   // code, etc. as named below
   char name[50];
   char city[50];
   char state[5];
   char country[30];
} Airport;
```

Next, in the main program we can define an array of airports:

```
Airport airportList[MAX_AIRPPRTS];
```

You can define MAX_AIRPORTS to be 4000 for the time being. Create a new file **task5.c** to have the following contents:

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

   Airport airportList[4000];
   int nA = 0;

   if (argc != 2) {
      printf("Cannot proceed: Missing file name.\n");
      exit(EXIT_FAILURE);
   }
   readData(argv[1], airportList, &nA);
   show(airportList[0]);

   return(0);
} // main()

void readData(char *file, Airport list[], int *n) {

   FILE *fp;

   int i = 0;      // # of airports read

   printf("Opening file: %s\n", file);
   fp = fopen(file, "r");
   if (fp == NULL) {
      printf("Unable to open file %s.\n", file);
      exit(EXIT_FAILURE);
   }
   printf("Opened file %s.\n", file);
```

```
   printf("Reading...");

   char line[80];
   fgets(line, sizeof(line), fp); // ignore first line
   while ( (fgets(line, sizeof(line), fp)) != NULL) {
      int l = strlen(line);
      line[l-1] = '\0';
      list[i] = parse(line);
      i++;
   } // reads and parses each line into an airport entry

   *n = i;
   fclose(fp);
   printf("...done. [%d]\n", i);
} // readData()
```

From what we learned in Task#3, we can write the `parse()` function as shown below:

```
Airport parse(char * str) { // Parse the string in str into Airport tokens

   Airport result;

   char *token;
   token = strtok(str, ",");
   strcpy(result.code, token);

   token = strtok(NULL, ",");
   strcpy(result.name, token);

   token = strtok(NULL, ",");
   strcpy(result.city, token);

   token = strtok(NULL, ",");
   strcpy(result.state, token);

   token = strtok(NULL, ",");
   strcpy(result.country, token);

   return result;
} // parse()
```

As before, complete the program, compile and run it. Note: the function `show()` prints out a given airport entry. Once completed, you should get the following output:

```
Opening file: code.txt
Opened file: code.txt
Reading……done. [3376]
00M – Thigpen
Bay Springs, MS (USA)
```

Notice that there are a total of 3376 airports.

Also, try this version:

The `parse()` function above, takes advantage of C's feature that allows C functions to return **struct** values as return values. As you can imagine, the **struct** can be defined to use large amounts of memory. The `Airport` **struct** uses exactly 140 bytes. This is not that big, so the way we have defined `parse()` is just fine. However, you should be aware that if structures get large, this use of `struct` return values can get very expensive. Instead, you can write `parse()` to either (1) take a reference to a **struct** as a parameter; or (2) return a pointer to a **struct**. The latter requires allocating memory within `parse()` to store the values in the **struct**. We will learn how to do this in a later class/lab. But passing a reference to a **struct** is easy and, as we learned in class, we can use the special syntax (`->`) to dereference **struct** variables. This is shown below:

```
void parse(char * str, Airport *a) {

   char *token;
   token = strtok(str, ",");
   strcpy(a->code, token);

   token = strtok(NULL, ",");
   strcpy(a->name, token);

   token = strtok(NULL, ",");
   strcpy(a->city, token);

   token = strtok(NULL, ",");
   strcpy(a->state, token);

   token = strtok(NULL, ",");
   strcpy(a->country, token);
} // parse()
```

This function should be called in `readData()` using the command:

```
parse(line, &list[i]);
```

If you have time in the lab, or afterwards, do try out this version. We will be making use of the features here in later programs.

Task#6: Finishing Up!

Once done, you can finish the program that searches the `airportList` list for a given airport code using linear search. See Assignment 5 for more details. This lab is due on Wednesday, November 8. Send an e-mail to your instructor when completed.