CSc 352 (Fall 2018): Assignment 8

Due Date: 11:59PM Sat, Nov 3

The purpose of this assignment is to do more involved work with pointers, linked lists, and memory allocation, to do a little work with command line arguments and reading from a file, and to learn to free all allocated memory.

General Requirements

1. Your C code should adhere to the coding standards for this class as listed in the Documents section on the Resources tab for Piazza. This includes protecting against buffer overflows whenever you read strings.
2. Your programs should indicate if they executed without any problems via their *exit status*, i.e., the value returned by the program when it terminates:

Execution	Exit Status
Normal, no problems	**0**
Error or problem encountered	**1**

3. Under *bash* you can check the exit status of a command or program ***cmd*** by typing the command "**echo $?**" immediately after the execution of ***cmd***. A program can exit with status *n* by executing "`exit(`*n*`)`" anywhere in the program, or by having `main()` execute the statement "`return(`*n*`)`".
4. Remember your code will be graded on lectura using a grading script. You should test your code on lectura using the `diff` command to compare your output to that of the example executable.
5. To get full points your code should compile without warnings or errors when the `-Wall` flag is set in `gcc`
6. Anytime you input a string you must protect against a buffer overflow. Review slides 82 – 87 of the basic_C deck.
7. You must check the return values to system calls that might fail due to not being able to allocate memory. (e.g. Check that malloc/calloc don't return NULL) getline() is an exception to this rule.
8. Your code must run without errors using valgrind.
9. **NEW REQUIREMENT, your program must free all allocated memory before exiting.**

Testing

Example executables of the programs will be made available. You should copy and run these programs on lectura to test your program's output and to answer questions you might have about how the program is supposed to operate. Our class has a home directory on lectura which is:

`/home/cs352/fall18`

You all have access to this directory. The example programs will always be in the appropriate **`assignments/assg#/prob#`** subdirectory of this directory. They will have the same name as the assigned program with "ex" added to the start and the capitalization changed to maintain camelback. So, for example, if the assigned program is **`theBigProgram`**, then the example executable will be named **`exTheBigProgram`**. You should use the appropriate UNIX commands to copy these executables to your own directory.

Your programs will be graded by a script. This will include a timeout for all test cases. There must be a timeout or programs that don't terminate will cause the grading script to never finish. This time out will never be less than 10 times the time it takes the example executable to complete with that test input and will usually be much longer than that. If your program takes an exceedingly long time to complete compared to the example code, you may want to think about how to clean up your implementation.

Makefiles

You will be required to include a Makefile with each program. Running the command:

`make` ***`progName`***

should create the executable file *progName*, where *progName* is the program name listed for the problem. The gcc command in your Makefile must include the **`-Wall`** flag. Other than that, the command may have any flags you desire.

Submission Instructions

Your solutions are to be turned in on the host **lectura.cs.arizona.edu**. Since the assignment will be graded by a script, it is important you have the directory structure and the names of the files exact. Remember that UNIX is case sensitive, so make sure the capitalization is also correct. For all our assignments the directory structure should be as follows: The root directory will be named **assg#**, where # is the number of the current assignment. Inside that directory should be a subdirectory for each problem. These directories will be named **prob#** where # is the number of the problem within the assignment. Inside these directories should be any files required by the problem descriptions. For this assignment the directory structure should look like:

```
assg8
   prob1
      calls.c
      Makefile
```

To submit your solutions, go to the directory containing your assg8 directory and use the following command:

turnin cs352f18-assg8 **assg8**

Problems

prob1: calls

It's time for some data mining! For this assignment you will write a C program in a file called **calls.c** and a **Makefile** which creates the executable **calls** that will open up a set of files (usually more than one), each of which has lines that are pairs of phone numbers (representing phone calls). You will record how many times each phone number has talked to each other phone number. After reading the files and creating a data structure, you will read queries from **stdin**. Each query will be a pair of phone numbers. If the two numbers have talked to each other, you will print how many times they talked (as indicated below). If they have not talked together, you will see if the numbers are connected through conversations. If they have, you will print the shortest path that connects the numbers. Details for this are below.

- **Invocation:**

  Your program will be invoked with one or more file names specified as command line arguments:

  calls *inFile*$_1$ **[** *inFile*$_i$ **]...**

  at least one inFile must be specified. It is followed by an optional number of additional files.

- **Input:**

  All input for this program will consist of lines of two different phone numbers, where the phone numbers are specified by whitespace. The phone numbers should have the format, ddd-ddd-dddd where each d is a digit (0-9). So all input lines should look like:

  ddd-ddd-dddd ddd-ddd-dddd

  where the phone numbers are not identical and are separated by one or more spaces or tabs. Blank lines (lines containing only whitespace) are also allowed and should be ignored. A line in any other format should cause and error message to be printed to stderr and the line should be ignored.

- **Program behavior:**

  Your program should open each of the specified input files for reading. If any of the files cannot be opened, you should print and error message to **stderr** and continue to the next given file. You will read the lines of phone numbers and use them to create a graph, much like you did in the last assignment. Here the phone numbers will be the vertices and the edges will represent a call between those numbers.

After all the files have been read, your program will read input from stdin. As with reading the files, it will read in lines, each of which should be a pair of phone numbers. These are queries. If the two numbers have talked your program will print out:

`Talked` *`n`* `times`

where n is the number of times the two numbers talked. Print this statement using:

```
printf("Talked %d times\n", num);
```

where num is an int variable containing the number of times the numbers talked.

If there was no direct call between the two numbers, then the program should check to see if they are connected indirectly. Two numbers are connected indirectly if there is a path of phone calls between them. For example suppose A did not talk to B, but A talked to C and C talked to B. In this case A is indirectly connected to B. If this is the case your program should print out the message:

`Connected through` *`n`* `numbers`

where n is the smallest number of phone numbers connecting the two numbers. Print this statement using the command:

```
printf("Connected through %d numbers\n", num);
```

where num is an int variable containing the smallest number of connecting numbers. The smallest number of connecting numbers is the number of phone numbers between the target phone numbers on the SHORTEST path connecting them. For example, if A talked to C and C talked to D, and D talked to B, then there are two connecting numbers on this path from A to B. However if A also talked to E and E talked to B, then there is only one connecting number on this path. In this case, the shorter path is A→E→B so the number 1 should be printed.

If there is no connecting path between the two numbers, you should print out the message:

```
Not connected
```

Note, **please be careful of the format of these outputs**. Check your output against the example executable using diff. I will use the –Z option on diff to grade these (meaning whitespace at the end of the line will be ignored), but I would hate for someone's program to calculate the values correctly but lose all the points because of extra punctuation or misspelling.

If during the input of queries a phone number is entered that is not in the graph this should also be treated as a nonfatal error. (an error message printed to stderr and the input line ignored) It is also a nonfatal error if the two phone numbers are the same.

- **Error Conditions:**

  Non-fatal errors: These should all be listed above. They include specifying an input file that can't be opened for reading or a nonblank line that contains something other than two different phone numbers in correct format. It is also an error if a phone number that is not in the graph is queried. Remember to exit with a status of 1 if any error is encountered.

  Fatal errors: If no input files are given as command line arguments. (Failure to get allocated memory is also a fatal error but not one that will be tested. Just make sure you check the return values of malloc, calloc, and strdup.)

- **Restrictions:**

  The usual restrictions apply to this program. You may not allocate space for a large array to save the data structure. Instead you must implement the graph using linked lists.

- **Data structure:**

  As in the last assignment, the data structure you should use for this program is an *adjacency list* to represent a graph. Note that in the last assignment the graph you created was directed, but in this assignment it should be undirected. If A talked to B, then B also talked to A. The easiest way to represent this is to add two edges, one from A and one from B, for every line you process.

  Also note that vertices are added implicitly in this program. By this I mean that we don't have a specific command to add phone numbers. Instead, if a line indicates that there was a call between A and B and A is not in the list of phone numbers it should be added. A last difference between this graph and the one you did for the last assignment is that you need to keep track of the number of calls made. To do this you will need to add a field to the struct you use to represent the graph's edges to keep track of how many calls were made.

- **Algorithm:**

  You need to find the shortest path connecting two nodes. For this a depth first search will not work. A depth first search finds if there is a path between two vertices, but not the shortest path. To find the shortest path, we will use a breadth first search. A depth first search recursively searches all the children of each vertex. A breadth first search puts all the children on a queue. This way all the children are searched before the grandchildren, etc. Here is the algorithm for a breadth first search:

```
//BFS does a breadth first search to find the shortest path
//from Start to Target. It returns the size of that path if there
//is a path from Start to Target and returns -1 otherwise
int BFS(Start, Target)
   mark Start as queued
   set level of Start to 0
   add Start to the queue

   while queue is not empty do
      Take A from top of queue
      If A = Target, return A.level
      For all children C of A do
         if C not queued
            mark C as queued
            set level of C to A.level + 1
            add C to queue

  return -1    //If you get through loop without finding Target,
               //there is no path from Start to Target
```

Note that in C you will have to implement this queue with a linked list. Don't forget to free it again after you're done using it if applicable.

Please note: the level this algorithm gives you is the distance between two vertices. The program asks for the number of vertices between the two vertices, so the number you will output is the value returned - 1.

- **Makefile**:

  In addition to your source file, you should submit a make file named **Makefile** that supports at least the following functionality:

  make calls
  Compiles the C source code to create an executable named **`linked`**. The compiler options used should include **`-Wall`**.

Don't forget that for this assignment you must free all your memory before exiting. Use valgrind to make sure all the memory is freed and that you have no memory errors.