CSc 352 (Fall 2023): Assignment 6

Due Date: 11:59PM Friday, Oct 20

The purpose of this assignment is to to work with linked lists, memory allocation, representing a graph in c, command line arguments, reading from a file, and optionally using free().

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 70 – 75 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 meaning that valgrind shows the line "ERROR SUMMARY: 0 errors" for all test cases.
9. **You may receive 20% extra credit if your program frees all allocated memory before exiting on all test cases.**

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/fall23`

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:

```
assg6
   prob1
      reach.c
      Makefile
```

To submit your solutions, go to the directory containing your assg5 directory and use the following command:

turnin cs352f23-assg6 assg6

Problems

prob1: reach

An important component of *safety analysis* of industrial systems is the determination of whether (and under what conditions) a system can enter a "bad" state. For example, for a traffic light we might want to know whether it is ever possible for the light to be green in both directions (this would be a bad state). This is an instance of the *reachability problem* or directed graphs: the general idea is to model the system in terms of a set of states and transitions between these states, and then checking whether certain bad states can be reached from some given initial state. This program involves writing code to solve this problem for any given (directed) input graph.

Write a C program, in a file **reach.c**, and a Makefile that creates the executable **reach** that behaves as described below:

- **Invocation:** Your program will be invoked with an *optional* command-line argument:

  ```
  reach   [ inFile ]
  ```

  where "`[ inFile ]`" indicates an optional argument that is the name of a file.

  If a command-line argument is specified, it should be the name of a readable file.

- **Input:** If an input file has been specified via a command-line argument, your program will take its input from this file. If no command-line argument is specified, it will read its input from **stdin**.

  The input will consist of a sequence of directives of the following form, **one per line**:

  op args

  where:

  - *op* specifies an operation, and can be one of: `@n`, `@e`, and `@q`; and
  - *args* is a sequence of zero or more arguments to the operation. The number of arguments depends on the operation (see below).
  - The *op* and *args* are separated by whitespace. (Hint: so it makes sense to use a form of scanf to parse the lines)

  Each line mentions one or more vertex names according to the format described below; you may assume that each such vertex name is of length at most 64 characters. (But as always, protect against buffer overflow.)

- **Program behavior:** Your program will read in input directives, one per line, and use them to construct a representation of a directed graph as specified by the input directives, until no more input can be read in. The program will compute and print out the appropriate reachability relations when so prompted by input directives.

  The program behaviors for different input directives are as follows:

  - `@n` *vName* : *vName* is an alphanumeric string of length at most 64. This declares *vName* as a vertex in the graph.

  - `@e` *vName1* *vName2* : *vName1* and *vName2* are previously-declared (using "`@n`") vertex names. This specifies that there is a directed edge from *vName1* to *vName2* in the graph.

    If there is already an edge from *vName1* to *vName2*, this directive has no effect (this is not an error).

  - `@q` *vName1* *vName2* : *vName1* and *vName2* are previously-declared vertex names. This asks whether there is a (directed) path from *vName1* to *vName2* in the graph. Your program should process the directed graph constructed up to that point to determine whether there is a path from *vName1* to *vName2*, and print out the result using the command

    `printf("%d\n", pathExists);` /* `pathExists` = 1 if there is such a path, 0 otherwise */

  Vertex names are case sensitive.

  If an input line does not adhere to the format described above, your program should give an appropriate error message, ignore the offending line, and continue processing. In this case the eventual return value of the program should indicate that an error occurred during processing.

- **Error Conditions:**

  Non-fatal errors: more than one command-line argument specified (ignore any additional arguments after the first one); input directives do not follow the format specified (ignore the offending lines), a node is declared more than once, an edge declaration contains a node not previously declared.

  Fatal errors: input file does not exist or is not readable.

- **Data structure:** The data structure you should use for this program is an *adjacency list*. To implement this you will use two structs. One to represent the nodes and the other to represent the edges between the nodes. Notice that for this problem the edges are *directed*: just because there is an edge from *A* to *B* does not mean there is an edge from *B*

to *A*. The nodes will be organized in a linked list. Each node will have an associated linked list of edges, one for each edge coming from the node. This looks very much like the data structure for the anagrams2 program. Here, the struct for the edges needs to contain the node the edge goes to, and a pointer to the next edge from the node. Students are always tempted to store the name of the node in this struct, but it is far faster and more efficient to store a pointer to the node instead.

- **Algorithm:** The simplest way to solve this problem is to use depth-first search. Pseudocode for this algorithm is as follows. Here, if there is an edge from a vertex *A* to a vertex *B*, we say that *B* is a *successor* of *A*.

```
/* dfs(fromNode, toNode) -- returns true if there is a directed
                            path from fromNode to toNode.
   To find whether there is a path from vertex A to vertex B:
     1) mark every vertex as "not visited"
     2) compute dfs(A, B)
*/

int dfs(fromNode, toNode) {
    if (fromNode == toNode) return 1;

    if (fromNode is marked "visited") return 0;

    mark fromNode as "visited";

    for each successor midNode of fromNode do {
        if (dfs(midNode, toNode)) return 1;
    }

    return 0;
}
```

- **Simple Example** (you will want to test on more complex graphs**)**

  The following input:

```
@n a1
@n a2
@e a1 a2
@q a1 a2
@n a3
@n a4
@e a2 a3
@q a1 a3
@q a1 a4
```

  Should produce the output

```
1
1
0
```