CSc 352 (Fall 2018): Assignment 7

Due Date: 11:59PM Sat, Oct 27

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

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. NOTE you do not need to free memory for this assignment, but you will get extra credit if you do. You need valgrind to say "ERROR SUMMARY: 0 errors" in order to get full credit.

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:

```
assg7
   prob1
      linked.c
      Makefile
```

To submit your solutions, go to the directory containing your assg7 directory and use the following command:

turnin cs352f18-assg7 **assg7**

Problems

prob1: linked

As you know web pages can contain links to other pages. Even if one page does not link directly to another page, you might still get from one page to another by following links on pages in between. For example, suppose page A links to page B and page B links to page C. You can get from page A to page C by following a chain (path) of links, first going to B and then going to C. You will write a program in a file called linked.c and a Makefile which creates from that source code a program called linked, that will use a graph to create a model of linked web pages, and answer queries about whether one page is connected to another through a chain of links.

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

  `linked   [ inFile ]`

  "**`[ 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: **`@addPages`**, **`@addLinks`**, and **`@isConnected`**; and
  - *args* is a sequence of zero or more arguments to the operation. The number of arguments depends on the operation (see below).

  Each line mentions one or more page names according to the format described below; you may assume that each such page name is of length at most 64 characters.

- **Program behavior:** Your program will read in commands, **one per line**, and use them to construct a representation of the pages and links and to answer queries about whether two pages are connected by a chain of links. You will read the input until end of file is reached, processing the commands one at a time as they are read. Note the data structure you will be using for this program is a directed graph. You may think of the pages as the vertices and the links as the edges.

The program behaviors for different input directives are as follows:

- **`@addPages`** *$Name_1$ $Name_2$ . . . $Name_n$*
  This the command followed by a list of n ($n \geq 0$) names of pages, separated by whitespace, to be added to the structure. A name can be any string, but you may assume no name is longer than 64 characters. If no names are listed (n = 0), this is a legal command but of course it does nothing. It is a non-fatal error if one or more of the names have already been added as pages. In this case, as for all nonfatal errors, an error message should be printed, but the other names on that line should still be added.

- **`@addLinks`** *sourcePage $Page_1$ $Page_2$ . . . $Page_n$*
  This is the command, followed by the name of the page containing the links, and then a list of n ($n \geq 0$) pages linked to. It is a nonfatal error if no sourcePage is given. If the sourcePage does not exist, that is also a nonfatal error and the command should be ignored. If a linked page does not exist, that is also a nonfatal error, but the rest of the links should still be added.

  It is NOT an error if a link already exists on the source page to some $Page_i$. You may ignore the instruction to add that link a second time. It is also not an error to include a link from a page to itself, although this also adds no information since all pages are connected to themselves (see below).

- **`@isConnected`** *$Page_1$ $Page_2$*
  This is the command followed by two page names. It asks whether there is a chain of links connected page *$Page_1$* to page *$Page_2$*. Your program should answer this question based on the pages and links processed up to that point. (i.e. based on the commands that came before this query, not looking yet at the commands that come after it) It should print out a 0 if the pages are not connected and a 1 if they are. Use something like:

  ```
  printf("%d\n", pathExists);  /* pathExists = 1 if there pages are conneced,
                                  0 otherwise */
  ```

  It is a nonFatal error if either of the pages don't exist, or if the string `@isConnected` is not followed by exactly two page names. In this case an error message should be printed to **stderr** and the command ignored.

  In the special case where *$Page_1$ = $Page_2$* we will print a 1. In other words, all pages are connected to themselves.

Blank lines (lines containing only whitespace) in the input are not errors and should be ignored. Any line of input whose first string is something other than `@addPages`, `@addLinks`, or `@isConnected` is a nonfatal error. An error message should be printed to **stderr** and the line ignored.

page names are case sensitive. (This is to make the program easier to write)

Remember each command is on a separate line. One that line all the arguments are separated by some positive number of tabs or spaces. This means you will need to read the input in a line at a time, but then you will have to parse each line. I suggest you use **sscanf()** to parse the lines, but for this assignment you may also use **strtok()** if you would like to.

- **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 (see descriptions above). For all nonfatal errors you should print an error message to **stderr**, but then continue processing. As you know by now, if you have encountered any error your exit status should be 1.

  Fatal errors: input file does not exist or is not readable. We call an error a fatal error if it means the program should halt when the error is encountered. In this case you print an error message to **stderr** and exit immediately with a status of 1.

- **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 pages and the other to represent the links between the pages. The pages will be organized in a linked list. Each page will have an associated linked list of links, one for each link coming from the page. This looks very much like the data structure for the noVowels2 program. Here, the struct for the edges needs to contain the page being linked to, and a pointer to the next link on the page. Students are always tempted to store the name of the page being linked to in this struct. Don't. It is far faster and more efficient to store a pointer to the struct representing the page instead.

- **Algorithm:** The simplest way to solve this problem is to use depth-first search. Pseudocode for this algorithm is as follows.

```
/* dfs(fromPage, toPage) -- returns true if there is a path from
                            fromPage to toPage.
       To find whether there is a path from page A to page B:
       1) mark every page as "not visited"
       2) compute dfs(A, B)
*/
int dfs(fromPage, toPage) {
   if (fromPage == toPage) return 1;

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

   mark fromPage as "visited";

   for each page midPage linked to by fromPage do {
```

```
        if (dfs(midPage, toPage)) return 1;
    }

    return 0;
}
```

- **Makefile**:

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

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

Extra credit [20%]: Free up all of the memory allocated by your program before it exits. The resulting code should (obviously) still generate the correct output and run cleanly under valgrind.