CSc 352 (Fall 2025): Assignment 9

Due Date: 11:59PM Fri, Nov 21

The purpose of this assignment is to do more involved work with pointers, linked lists, memory allocation, command line arguments, reading from a file, using free(), representing graphs, and learning about how make works.

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.
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. Your program must free all allocated memory before exiting.
10. **NEW REQUIREMENT: You must break your code up into at least two source files (.c) and one header (.h) file. Your Makefile should create the executable file in a way that only the files that need to be recompiled are recompiled.**

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/fall25
```

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
or
make progName
```

should create the executable file *progName*, where *progName* is the program name listed for the problem. The gcc commands that compile the ob 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:

```
assg9
   prob1
      <files needed to make project>
```

To submit your solutions, go to the directory containing your assg9 directory and use the following command:

turnin cs352f25-assg9 **assg9**

prob1: mymake (part1)

This problem involves writing a C program that implements part the core functionality of the *make* utility. This assignment involves reading in a file specifying dependencies and rules as in *make* (though with a different and more explicit syntax); constructing a dependency tree; and then traversing the tree starting with some specified target. The extension to check timestamps and execute the associated commands where necessary will be part of a future assignment.

- **Invocation:** Your program will be compiled to an executable named **mymake** . It will be invoked as follows:

  mymake *aMakeFile* *aTarget*

  aMakeFile is a file specifying dependencies and rules according to the format given below; and *aTarget* is the name of a target appearing in *aMakeFile*.

- **Behavior:** An invocation "`mymake` *aMakeFile aTarget*" of your program should behave as follows: (i) read in the targets and dependencies specified in *aMakeFile* and construct the corresponding data dependency graph; (ii) traverse this graph using a *postorder traversal*, starting at the node corresponding to the target *aTarget*. (The Wikipedia discusses the general notion of tree traversals, including postorder traversals, in more detail.)

- **Files:** You should structure your program so that conceptually distinct pieces of the program reside in distinct files. For instance, the code for reading the makefile specifications might be in a different file from the functions dealing with graphs. (for example, finding a node, adding a node, traversing the graph, etc.) You should include at least 2 different files of source code and one header file. You can include more if you feel so inclined.

- **Output:** The output produced by your program is a sequence of node names obtained from the postorder traversal of the dependency graph (see below) followed by all the commands for those nodes with the commands indented two spaces. To guarantee your output matches the example code, visit the children of a node in the order they are listed in the make file.

- **Errors:** Error messages should be sensible and informative (use **perror** where necessary) and should be sent to **stderr**.

  The following are all fatal errors and should cause the program to exit immediately with exit status **1**.

  - An input file or a target is not specified.
  - Too many arguments are specified.
  - The input file cannot be opened for reading.

- The input file is in an illegal format.
- The specified target is not defined in the input file.

Assumptions and Restrictions

- **Assumptions:** You may assume that a *word* in the input mymake file has length at most 64, <u>but you must still protect against buffer overflow</u>! Also, there is no limit on the length of command lines or the number of them. Your program should all for this with dynamic memory allocation for each command line and a linked list for the list of different commands.

- **Restrictions:** The point of this exercise is to work with pointers and dynamic data structures. Programs that use statically preallocated memory for the primary structures will not get credit. You will also not get credit for solutions that use **realloc** or which simulate **realloc** by making repeated calls to **malloc** or **calloc**. (In other words, use linked lists as in prior programs.)

mymake makefile specifications

Terminology

- A *<u>whitespace character</u>* is any character for which the function **isspace()** returns a non-zero value, e.g., blanks, tabs, newlines. Note that since rules are on lines that you should read with getline(), we can essentially ignore the possibility of newlines in what follows.

- A *<u>word</u>* is a nonempty sequence of characters that are not whitespace and not :.

File Structure

A "mymake file" consists of a sequence of *rules*. These have the structure specified below.

1. Rules

A rule consists of a single *target specification* followed by a list of zero or more *commands*. The commands are listed on separate lines. These have the structure specified below.

1.1. Target Specification

A target specification is of the form

`target` : str_1 str_2 . . . str_n

There can be any number of dependencies (strings separated by whitespace after the ":", including none). There can be any amount of whitespace before and after the ":", including none. **Important**: A word cannot appear as the target for more than one rule.

1.2. Commands

A command is of the form

```
\t cmd
```

where *cmd* is a bash command. The tab character, \t, must be exactly the first character on the line. The space after the \t is just for illustration purposes. You can choose to support leading spaces before *cmd*, but we will not test that.

2. Examples

The following are examples of mymake files. (Note that in these examples by "\t" I mean a tab character, not the two chars '\' and 't'):

Example 1: The following is legal:

```
spellcheck.o : utils.h   spellcheck.h   spellcheck.c
\t gcc -Wall -c spellcheck.c

hash.o :hash.c utils.h hash.h
\t gcc -Wall hash.c

spellcheck    :      hash.o     spellcheck.o
\t gcc *.o -o spellcheck

testStuff:
\t spellcheck <testFile1 >out1
\t spellcheck <testFile2 >out2
\t spellcheck <testFile3 >out3
```

Example 2: The following is illegal. Because **spellcheck** appears as a target more than once.

```
spellcheck : hash.c spellcheck.c
\t gcc hash.c spellcheck.c -o spellcheck spellcheck.o : utils.h   spellcheck.h   spellcheck.c
\t gcc -Wall -c spellcheck.c

hash.o : hash.c utils.h hash.h
\t gcc -Wall hash.c

spellcheck : hash.o spellcheck.o
\t gcc *.o -o spellcheck
```

Dependency Graphs

A *dependency graph* is a data structure that captures the dependencies between different files as specified in a make file. To keep this discussion specific, we will focus here on rules in mymake format; however, the concepts are not specific to this assignment and generalize readily to the full **make** utility.

1. Structure

A dependency graph is a *directed graph* satisfying the following:

Each node represents a *target* or something a target depends on as given in the input mymake file. Thus, for each rule of the form

```
A :  . . .  B . . .
\t  cmd1
\t  cmd2
```

there is a node named *A* and a node named *B* in the dependency graph and there is edge from node *A* to node *B* if *A* "*depends on*" *B*.

The sequence of commands (cmd1 and cmd2 in the above example) is associated with the dependency graph node for the target for the rule.

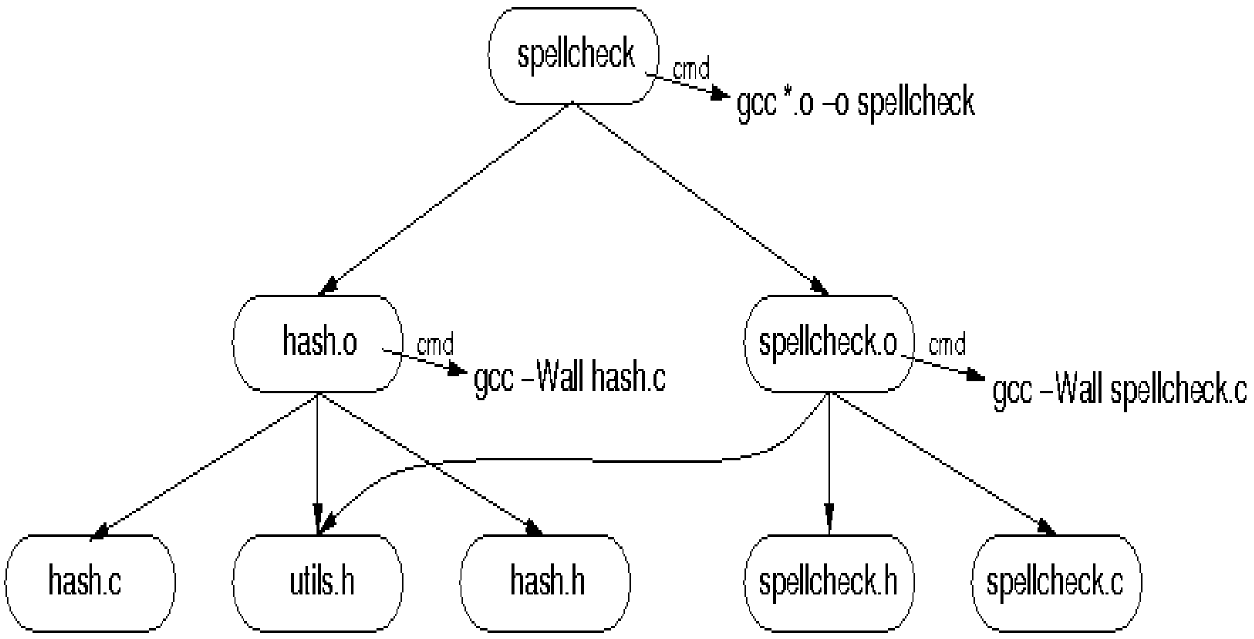
The following example illustrates the notion of dependency graphs. Consider the following mymake file:

```
spellcheck.o : utils.h   spellcheck.h   spellcheck.c
\t   gcc -Wall -c spellcheck.c

hash.o : hash.c utils.h hash.h
\t   gcc -Wall hash.c

spellcheck : hash.o spellcheck.o
\t   gcc *.o -o spellcheck
```

The corresponding dependency graph is as follows:



The ordering on the children of each node in a dependency graph is significant: it reflects the left-to-right ordering on the dependencies specified as part of a rule. For example, the first rule in the example above gives the dependencies of the target "**`spellcheck.o`**" in the following left-to-right order:

```
utils.h
spellcheck.h
spellcheck.c
```

The children of the node corresponding to this target in the dependency graph shown above reflect this ordering.

2. Post-order Traversal

Let the sequence of children of the node *aTarget* be <$aChild_1$, $aChild_2$, . . . , $aChild_n$>, where the ordering on the nodes $aChild_i$ reflects as the ordering specified in the target specification for the rule for *aTarget*, as discussed above. The postorder traversal of the graph starting at node *aTarget* is carried out as follows:

```
PostOrder (aTarget)
   If aTarget visited return
   mark aTarget as visited
   for i = 1 to n (in that order)
       PostOrder(aChild_i);
   Process aTarget
```

For the purposes of this assignment, "*processing a node*" simply means printing out its name, without any extraneous whitespace, followed by a newline, and then each command (if any) associated with that target. Each command should be on its own line and should be indented two spaces.

This is essentially a depth first search

Example: A post-order traversal of the dependency graph shown above, starting at the root node `spellcheck`, produces the following:

```
hash.c
utils.h
hash.h
hash.o
  gcc -Wall hash.c
spellcheck.h
spellcheck.c
spellcheck.o
  gcc -Wall -c spellcheck.c
spellcheck
  gcc *.o -o spellcheck
```