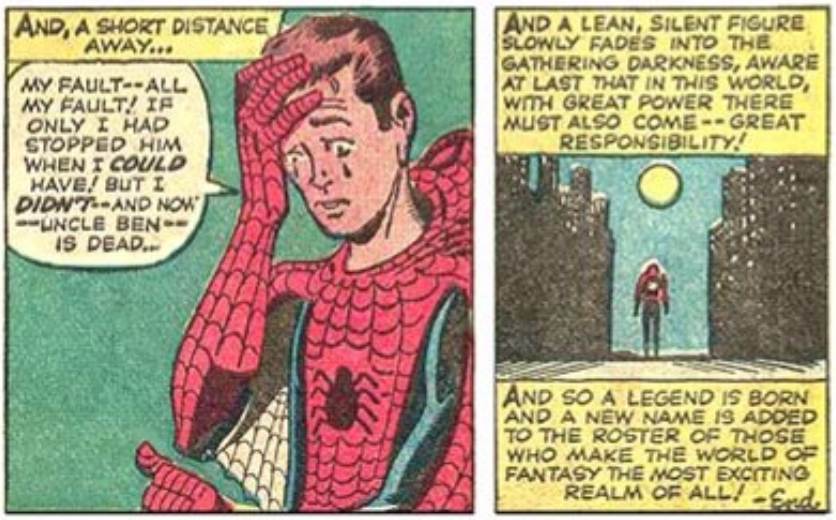
CS 105 (C++)

Assignment 4: Dynamically Allocated Stack



I. Overview

In this assignment, you will implement a dynamically allocated stack.

Your stack will be driven by commands from an input file, and when finished, your program will write the final stack to an output file.

The following are some important details of how your program must be implemented.

- Your stack will consist of a linked list of dynamically allocated nodes, with the head of the list being the top of the stack.
- You will maintain a global pointer to the head/top of the stack.
- Nodes will be defined as structs, with each node containing a data element of type `char` and a pointer to the next node.
- The last node (or the head, if the stack is empty) must point to `NULL`.
- You must define the following functions to operate on your stack:

    - `void push(char c) /* push c onto the top of the stack */`
    - `char pop() /* pop the value off the top of the stack and return it */`
    - `int empty() /* return nonzero if the stack is empty, zero otherwise */`

- Your program will receive the names of its input and output files as its first and second arguments, respectively.
- The input file will consist of a sequence of stack commands, one on each line, to either push a character onto the stack or pop a value off of the stack. For example, to push `'a'` onto the stack, then pop it off,

the following lines would appear in the input file:

```
push a
pop
```

- When you have processed all of the input commands, print the final state of the stack (starting from the top) to the output file on a single line, with no spaces. (This is all that the output file should contain.)
- Although you can generally assume valid input and don't have to check for other errors, if an attempt is made to pop while the stack is empty, you must print an error message on `stderr` (**not** on `stdout`) and call `exit` (defined in `<stdlib.h>`).
- When you think your program is working properly, run it on the input file a4_test.txt and check the results. (I won't tell you ahead of time what the output should be, but if you get it right, I think you'll know it.)

Note: In this assignment you will be allocating and deallocating memory by hand. Design and implement your code carefully to ensure that this is done correctly. (Despite years of experience in this area, I still literally draw pictures of the memory management whenever I have to implement this kind of code.) During grading, I will test your memory handling using a program called "valgrind" as described in Section II below. You should run the same test yourself before submitting your work.

II. Grading

Note: In light of the importance of memory management and the danger of getting it wrong, improper allocation and deallocation will be counted in the (Provisional) Dealbreakers section of this assignment's grading.

- Minimum Requirements
  - Proper stack operation.
  - Proper use of input and output files.
  - Your work must be submitted in a single file called `main.c` .
  - This file must compile on a department UNIX machine with the following command:
    `cc main.c -o a4`
  - Before evaluation, your code must be submitted via `turnin`, using the following command on a department UNIX machine:
    `turnin --submit dlessin a4 main.c`
- Graded Elements
  - Proper use of `fscanf`.
  - Proper use of `fprintf`.
  - Proper use of `strcmp`.
  - Message on `stderr` and call to `exit` if pop on empty stack.
  - Proper use of `fclose`.
- Provisional Dealbreakers
  - Due to the importance of proper handling of dynamic memory, you will lose **50%** of your programming assignment grade if a valgrind evaluation of your work shows any memory leaks or errors.
  - **However**, in recognition of the difficulty of this task, you may resubmit your work at any time

before the end of the course for regrading on this portion of your score. (And of course, I'll be happy to help you find and fix any problems.)

- Valgrind evaluation is performed as follows:
  - Prepare your executable for valgrind by compiling with debugging information on and optimization off. Use the following command for this (Note that "`O0`" is a capital letter "o" followed by the number zero.):

    ```
    cc -g -O0 main.c -o a4
    ```
  - Run valgrind by prepending (i.e., adding at the beginning) the following to your normal command (including all normal arguments):

    ```
    valgrind --leak-check=yes
    ```

III. The More You Know

The following are some additional items that may be very important for you to know about this assignment.

- Due to an unusual coincidence, when running your program for this assignment, you should always run it as "./a4". This will avoid conflicts with a program/machine on the system which happens to have the name "a4".
- What do you do if your program crashes and you don't know why? Adding statements to print out diagnostic information as you go can often be quite useful, but if you're ready to learn a real debugger for C/C++, look up gdb (the GNU Project debugger).
- The following images illustrate the memory handling during the push and pop operations you'll be implementing for this assignment. Note that there are two cases shown for push and two cases shown for pop, HOWEVER, for this assignment it should be easiest to just write one push to handle both push cases and one pop to handle both pop cases.
  a4_stack_push_empty.png
  a4_stack_push_general.png
  a4_stack_pop_general.png
  a4_stack_pop_last.png
