CSE 303, Spring 2005, Assignment 5B (Counter)
Due: Tuesday 17 May, 9:00AM

Last updated: May 11

Summary: You will write a C file that (assuming several functions are provided in other files), computes the "probabilities" of three-word sequences in a text file. You will also write unit tests for the code you write. The sample solution, not including unit tests and the assumed declarations (see next paragraph) is about 50 lines.

Your code should include the following declarations for types and functions defined elsewhere:

```
typedef struct InputInfo * input_info_t;
input_info_t initialize_input(char*); // constructor
char * next_word(input_info_t); // getter
void complete_input(input_info_t); // destructor

typedef struct WordCounts * word_counts_t;
word_counts_t new_word_counts(); // constructor
void enter_word(word_counts_t,char*); // setter
void enter_word_pair(word_counts_t,char*,char*); // setter
void enter_word_triple(word_counts_t,char*,char*,char*); // setter
int get_word_count(word_counts_t,char*); // getter
int get_word_pair_count(word_counts_t,char*,char*); // getter
int get_word_triple_count(word_counts_t,char*,char*,char*); // getter
```

The first group defines a class-like interface for getting words from an input file. The constructor takes a filename. The getter returns a heap-allocated string containing (a copy of) the next word in the file (or `NULL` if there are no more words). Your code should free the space when you are done with the string. The destructor frees the space for the `struct InputInfo` "object"; you should call it exactly once after you are done calling `next_word` on the "object".

The second group defines a class-like interface for counting one-word, two-word, and three-word "phrases". To increment the count for a phrase (initially 0 implicitly), call the correct setter function (where the "pair" and "triple" versions take the words in the phrase in left-to-right order). To get a phrase's current count, call the appropriate getter method. Alas, the interface has no way to free the space associated with a `struct WordCounts` "object", so you do not have to worry about it.

1. Include the following type definition:

   ```
   struct WordInfo {
     ...
   };
   ```

   The fields of `struct WordInfo` should include one field of type `word_counts_t` (for counting phrases) and one of type `int` (for counting the total number of words encountered).

2. Implement the function `struct WordInfo * make_data(char * filename)`. The function returns a new heap-allocated `struct WordInfo *` that describes the occurrences of all one-word, two-word, and three-word "phrases" in the text file `filename`. Use the `input_info_t` "interface" for getting words. If the file has fewer than 3 words, given an error message to that effect and exit the program. Else count how many words are in the file and use a "word count object" to count how many times each one-word, two-word, and three-word phrase occurs. Make sure you free strings when you are done with them (the "word count object" does *not* keep copies of the pointers it is passed).

   Example: A 5-word file would has a total of 5 1-word phrases, 4 2-word phrases, and 3 3-word phrases. For example, in "b a a a b", the phrase counts are: (a, 3), (b, 2), (b a, 1), (a a, 2), (a b 1), (b a a, 1), (a a a, 1), and (a a b, 1).

3. Implement the function

   ```
   void get_probabilities(double * ans, struct WordInfo * w,
                               char * first, char * second, char * third)
   ```

   This function computes three "estimations" of the probability that a three-word phrase drawn randomly from the text described by `w` is the three-word phrase `first second third` (i.e., the strings in these arguments in left-to-right order). It stores the 3 estimations in `ans[0]`, `ans[1]`, and `ans[2]`. (You should assume the array `ans` points to is long enough.)

   - The first estimation is the "independent word model": The probability is $(w_1 \cdot w_2 \cdot w_3)/t^3$ where w_1 is the number of times the first word appears, w_2 the number of times the second word appears, w_3 the number of times the third word appears, and t is the total number of words.
   - The second estimation is the "independent pair model": The probability is $(p_{12} \cdot p_{23})/(t \cdot w_2)$ where p_{12} is the number of times the two-word phrase `first second` appears, p_{23} is the number of times the two-word phrase `second third` appears, and the other variables are defined above.
   - The third estimation is the "exact model": The probability is p_{123}/t where p_{123} is the number of times the three-word phrase `first second third` appears, and the other variables are defined above.

   Note: We are assuming that t is large enough that the difference between t and $t-2$ is insignificant.

4. Write *unit tests* for the functions above. Include comments indicating what different tests accomplish and a `main` function that runs your tests.

Extra Credit: Write functions `make_data_a` and `get_probablities_a` that are like `make_data` and `get_probablities` except that they consider any words that are *anagrams* of each other (the same number of each letter; just rearranged) to be the *same* word. Use helper functions so that you do not have "copied and pasted" code in your file.

Assessment: Your solutions should be:

- Correct C programs that compile without warnings using `gcc -Wall`.
- In good style, including indentation and line breaks
- Of reasonable size

Turn-in Instructions:

- Follow the link on the course website and follow the instructions there.
- **Problems 1–3 should have solutions in `counter.h` and `counter.c`; your unit tests should be in `counter_test.c`.**
- We should be able to compile your program via `gcc counter.c counter_test.c`, assuming `counter.h` is in the same directory.