Document Manager **Due date posted on Piazza**

1 Overview

In this project you implement a document manager program. It allows us to add paragraphs, lines to paragraphs, replace text, and edit a document. The next project relies on the code you implement for this project.

The objective is to practice C structures and string manipulation.

2 Grading Criteria

Your project grade will be determined as follows:

Results of public tests	44%
Results of release tests	24%
Results of secret tests	22%
Code style grading	10%

2.1 Good Faith Attempt

Remember that you need to satisfy the good faith attempt for every project in order to pass the class (see syllabus). The good faith attempt information for this project (e.g., requirements and deadline) will be posted on the class web page later on.

2.2 Obtaining project files

Copy the folder project2 available in the 216public projects directory to your 216 directory. This project description can be found in the 216public project_descriptions directory.

The project files include the following three files: (a) `document.h`, which provides prototypes for the functions you must implement; (b) `.submit`, needed to submit the project to the submit server; and a (c) `Makefile`, which will help you work on your project more efficiently.

2.3 Using Makefile

Makefiles will be covered more in-depth in class later. For now, think of a makefile as containing a list of commands for the shell to execute, thereby saving you from entering those commands individually. For example, "`gcc public01.c document.c`" creates the executable corresponding to the first public test. Equivalently, you can type "`make public01`" to create the executable corresponding to the first public test; the executable file will be named "public01". You can type "`make`" to create executables for all public tests. You can remove files that end with .o and executables (including a.out) by typing "`make clean`".

3 Specifications

The structure of a document is defined in document.h. Briefly, a document consists of a name and zero to `MAX_PARAGRAPHS` paragraphs. The name is a nul-terminated string of 1 to `MAX_STR_SIZE` characters. A paragraph consists of zero to `MAX_PARAGRAPH_LINES` lines. Each line is a nul-terminated string of 1 to `MAX_STR_SIZE` characters.

3.1 Functions

You must implement the functions described below. Instead of 0 and -1, use the macros SUCCESS and FAILURE defined in document.h. In these functions, assume as usual that a non-NULL pointer parameter points to a memory area of appropriate size; for example, a non-NULL `doc` parameter points to an area that can hold a Document struct.

1. `int init_document(Document *doc, const char *name)`

   Initializes `*doc` to have 0 paragraphs and sets the document's name based on the provided parameter value. The function returns FAILURE if doc is NULL, name is NULL, or if the length of the name is not between 1 and `MAX_STR_SIZE`; otherwise the function returns SUCCESS.

2. `int reset_document(Document *doc)`

   Sets the number of paragraphs to 0. The function returns FAILURE if doc is NULL; otherwise the function returns SUCCESS.

3. `int print_document(Document *doc)`

   Prints the document's name, number of paragraphs, followed by the paragraphs. Exactly one **empty line** (with zero characters before the newline) is inserted between every two successive paragraphs. The function returns FAILURE if doc is NULL; otherwise the function returns SUCCESS.

   The following illustrates an example of printing a document whose title is "Exercise Description" and has two paragraphs, each with three lines:

   ```
   Document name: "Exercise Description"
   Number of Paragraphs: 2
   First paragraph, First line
   First paragraph, Second line
   First paragraph, Third line

   Second paragraph, First line
   Second paragraph, Second line
   Second paragraph, Third line
   ```

   Note that the empty line is printed as long the document has two paragraphs, even if one or both of the paragraphs have zero lines.

4. `int add_paragraph_after(Document *doc, int paragraph_number)`

   Adds a paragraph after the specified paragraph number. Paragraph numbers in the document start at 1. If paragraph_number is 0 the paragraph is added at the beginning of the document. The function returns FAILURE if doc is NULL, the document already has the maximum number of paragraphs (MAX_PARAGRAPHS), or if paragraph_number is negative or higher than the number of paragraphs in the document; otherwise, the function returns SUCCESS.

5. `int add_line_after(Document *doc, int paragraph_number, int line_number, const char *new_line)`

   Adds a new line after the line with the specified line number. Line numbers start at 1. If line_number is 0, the new line is added at the beginning of the paragraph. The function returns FAILURE if doc is NULL, paragraph_number does not refer to an existing paragraph, the paragraph already has the maximum number of lines allowed, line_number is negative or higher than the available number of lines, or new_line is NULL; otherwise, the function returns SUCCESS. Assume that a non-NULL new_line is a nul-terminated string of length between 1 and `MAX_STR_SIZE`.

6. `int get_number_lines_paragraph(Document *doc, int paragraph_number, int *number_of_lines)`

   Returns the number of lines in a paragraph using the number_of_lines out parameter. The function returns FAILURE if doc or number_of_lines is NULL or if paragraph_number does not refer to an existing paragraph; otherwise, the function returns SUCCESS.

7. `int append_line(Document *doc, int paragraph_number, const char *new_line)`

   Appends a new line to the specified paragraph. The conditions that make add_line_after fail apply to this function as well. The function returns SUCCESS if the line is appended. Assume that a non-NULL new_line is a nul-terminated string of length between 1 and `MAX_STR_SIZE`.

8. `int remove_line(Document *doc, int paragraph_number, int line_number)`

   Removes the specified line from the paragraph. The function returns FAILURE if doc is NULL, paragraph_number does not refer to an existing paragraph, or line_number does not refer to an existing line; otherwise the function returns SUCCESS.

9. `int load_document(Document *doc, char data[][MAX_STR_SIZE + 1], int data_lines)`

   Adds the first data_lines number of rows from the data array to the document, starting a new paragraph at the beginning of the document. A row with an empty string starts a new paragraph. So if there are n empty rows in the first data_lines, the function adds $n + 1$ new paragraphs to the document.

   Assume that data_lines is non-negative, and if it is positive then each of the corresponding rows in the data array has a null-terminated string of length at most `MAX_STR_SIZE`. The function returns FAILURE if doc is NULL, data is NULL or data_lines is 0. It also returns FAILURE if adding data_lines number of rows would violate the max number of allowed paragraphs or the max number of lines in a paragraph; in this case, either leave the document unchanged or grow the document to the maximum allowed. In all other cases, the function returns SUCCESS.

10. `int replace_text(Document *doc, const char *target, const char *replacement)`

    Replaces every appearance of the text **target** in the document with the text **replacement**. Assume that the target is not the empty string. Assume that the replacement does not generate a line that exceeds the maximum allowed line length. The function returns FAILURE if doc, target or replacement is NULL; otherwise the function returns SUCCESS.

11. `int highlight_text(Document *doc, const char *target)`

    Highlights every appearance of the text **target** in the document, by surrounding the text with the strings HIGHLIGHT_START_STR and HIGHLIGHT_END_STR (see document.h). Assume that the target is not the empty string. Assume the highlighting does not cause any line to exceed the maximum allowed line length. The function returns FAILURE if doc or target is NULL; otherwise the function returns SUCCESS.

12. `int remove_text(Document *doc, const char *target)`

    Removes every appearance of the text **target** in the document. Assume the target is not the empty string. The function returns FAILURE if doc or target is NULL; otherwise the function returns SUCCESS.

3.2 Important Points and Hints

1. Do not modify document.h. Use the symbolic constants defined there (eg, `MAX_PARAGRAPHS_LINES`); our tests may use different values for these constants.
2. You must create a file named document.c that includes the file document.h.
3. Do not add a main() function to document.c. The main() function is provided by a driver file (e.g., public01.c). Think of document.c as a library.
4. IMPORTANT: Do not use dynamic memory allocation functions (`malloc()`, `calloc()`, etc). If you do, you will lose all the points associated with public, release, and secret tests.
5. If your code is not working properly, read the debugging guide available at:

   http://www.cs.umd.edu/~nelson/classes/resources/cdebugging/

6. Run valgrind as you develop your code. It will help you identify memory problems. Without using valgrind, you may not pass the secret tests.
7. String manipulation can become messy in this project. Break down your implementation (e.g., rely on auxiliary functions) so you can control the code complexity.
8. You can implement the replace text abstraction in different ways. For example, you can create the replacement text first and then replace the original, instead of directly editing the original.

9. Do not use strtok, strtok_r, strspn nor strcspn.
10. The next project will depend on the code you develop for this project.
11. Note that when you remove a line, you overwrite it with the following line (if any). If the last line is removed, there is no overwriting (but you do have to adjust the number of lines).
12. Recall that load_document puts its data at the beginning of the document. Thus if several load_document calls are issued to a document, the loaded data will appear in reverse order of the calls.
13. If we remove the only line from a paragraph, the paragraph does not disappear. The paragraph now have 0 lines.
14. If replace_text(&doc, "app", "ap") acts on the following line:
    ```
    is app131.  This course will be
    ```
    the result is
    ```
    is ap131.  This course will be
    ```
15. There is no need to initialize the lines array when a paragraph is added. There will be garbage in the lines array, but that is OK. If you were to print the document, only paragraphs with a number of lines different than 0 will be printed.
16. If there is nothing to compile or link when you execute "make", you will see something like: **make: Nothing to be done for 'all'.**
17. To create your own tests:
    a. Make a copy of one of the public tests (e.g., cp public01.c my_test.c)
    b. Add your code to my_test.c. Do not add a main() function to document.c.
    c. Compile (gcc my_test.c document.c)
18. A common mistake is to implement the replace_text function without using any auxiliary function (everything is done in this single function). You should have an auxiliary function (e.g., replace_in_line) that replaces text within a line. Once you have developed this function (an tested it), you can use it for replacing text across the whole document. Also, don't make the task harder than it needs to be. If you need to replace text in a given string, you could modify the original string, but that may be hard. It is better to have a second string variable that represents the result of replacing text. This second string variable will receive the characters that represent the replacement.
19. Both init_document and reset_document reset a document. The only difference is that one allows you to change the document's name.
20. The replace_text function is case sensitive. For example, if looking for "cat" it will not match "Cat".
21. Make sure that you add your own tests so you can test cases that might be covered by secret tests. It is imperative that you use valgrind to verify that your code works with your own tests.
22. For your code to compile in the submit server, you need to have an implementation for all the functions. If you have not finished/started a function implementation, provide an empty body as the implementation (returning a dummy value if a returned value is required).

3.3 Style grading

For this project, your code is expected to conform to the following style guidelines:

- Your code must have a comment at the beginning with your name, university ID number, and UMD Directory ID (i.e., your username on Grace).
- Follow the C style guidelines available at:

  http://www.cs.umd.edu/~nelson/classes/resources/cstyleguide/

4 Submission

4.1 Deliverables

The only file we will grade is document.c. We will use our versions of all header files to build our tests, so do not make any changes to the header files.

4.2 Procedure

To submit your project, execute the **submit** command in your project directory (project2).

5 Academic integrity statement

Please **carefully read** the academic honesty section of the course syllabus. We take academic integrity matters seriously. Please do not post assignment solutions online (e.g., Chegg, github) where others can see your work. This can lead to you being reported to the Office of Student Conduct.