CMSC 216 Project #2 Summer 2023

Document Manager **Due: Wed, Jun 21, 11:55 pm**

1 Overview

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

2 Objectives

To practice C structures and string manipulation.

You must implement this project individually. You may not work with other students and you may not discuss the project with other students. If you have project questions, post them in Piazza, or stop by during TAs' office hours.

You are responsible for verifying your project works on the submit server. If your project works in grace but not on the submit server, you will lose most (if not all) of the project points. You should submit often and verify the output your code is generating on the submit server.

3 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. Posting code online can lead to an academic case where you will be reported to the Office of Student Conduct.

This project has been used in the past and you may find implementations online. Notice we are aware of the code sources, so you will be part of an academic integrity case if you use any such sources (even if you modify them). If you violate academic integrity rules, we will ask for an XF in the course; no exceptions.

4 Grading Criteria

Your project grade will be determined as follows:

Results of public tests	45%
Results of release tests	25%
Results of secret tests	24%
Code style grading	6%

5 Project files

To obtain the project files copy the folder project2 available in the 216 public directory to your 216 directory.

We have supplied three files for your use in this project: (a) `document.h`, which provides prototypes for the functions you must implement; (b) `.submit`, which is necessary to submit the project to the submit server; and a (c) `Makefile`, which will help you work on your project more efficiently.

6 Specifications

6.1 Use of a Makefile

Makefiles will be covered more in-depth in class, but instead of issuing `gcc` commands every time you want to recompile your program, you can simply execute "`make`" from the command line. Make recompiles the public tests provided. Without the makefile you can compile a public test by compiling the test (e.g., public01.c)

and document.c. For example, gcc public01.c document.c will create the executable corresponding to the first public test. If you type ”make public01” you will create an executable that corresponds to the first public test; the executable name will be public01. You can remove files that end with .o and executables (including a.out) by typing ”make clean”. Typing ”make” will create executables for all public tests.

6.2 Functions

You must implement the functions described below. Instead of 0 and -1 you should use the macros SUCCESS and FAILURE defined in document.h.

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

   Initializes the document to be empty. An empty document is one with 0 paragraphs. The function will initialize the document's name based on the provided parameter value. The function will return FAILURE if doc is NULL, name is NULL, or if the length of the name provided exceeds `MAX_STR_SIZE`; otherwise the function will return 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 will return SUCCESS.

3. `int print_document(Document *doc)`

   Prints the document's name, number of paragraphs, followed by the paragraphs. Each paragraph is separated by a blank line. The following illustrates an example of printing a document whose title is ”Exercise Description” and has two paragraphs:

   ```
   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
   ```

   The function returns FAILURE if doc is NULL; otherwise the function will return SUCCESS.

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

   Adds a paragraph after the specified paragraph number. Paragraph numbers start at 1. If paragraph_number is 0 the paragraph will be added at the beginning of the document. The function will return FAILURE if doc is NULL, the document has the maximum number of paragraphs allowed (MAX_PARAGRAPHS) or if the paragraph_number is larger than the number of paragraphs available; otherwise, the function will return 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 the line number is 0, the line will be added at the beginning of the paragraph. The function will return FAILURE if doc is NULL, the paragraph_number exceeds the number of paragraphs available, the paragraph already has the maximum number of lines allowed, the line_number is larger than the available number of lines or if new_line is NULL; otherwise, the function will return SUCCESS.

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 will return FAILURE if doc or number_of_lines is NULL or if the paragraph_number is larger than the number of paragraphs available; otherwise, the function will return SUCCESS.

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

   Appends a line to the specified paragraph. The conditions that make the add_line_after fail apply to this function as well. The function will return SUCCESS if the line is appended.

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

   Removes the specified line from the paragraph. The function will return FAILURE if doc is NULL, the paragraph_number exceeds the number of paragraphs available or line_number is larger than the number of lines in the paragraph; otherwise the function will return SUCCESS.

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

   The function will add the first data_lines number of lines from the data array to the document. An empty string in the array will create a new paragraph. Notice that by default the function will create the first paragraph. You can assume that if data_lines is different than 0, the appropriate number of lines will be present in the data array. The function will return FAILURE if doc is NULL, data is NULL or data_lines is 0; otherwise the function will return SUCCESS.

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

    The function will replace the text **target** with the text **replacement** everywhere it appears in the document. You can assume the replacement will not generate a line that exceeds the maximum line length; also you can assume the target will not be the empty string.

    The function will return FAILURE if doc, target or replacement are NULL; otherwise the function will return SUCCESS.

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

    The function will highlight the text associated with **target** everywhere it appears in the document by surrounding the text with the strings HIGHLIGHT_START_STR and HIGHLIGHT_END_STR (see document.h). You can assume the highlighting will not exceed the maximum line length; also you can assume the target will not be the empty string. The function will return FAILURE if doc or target are NULL; otherwise the function will return SUCCESS.

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

    The function will remove the text **target** everywhere it appears in the document. You can assume the target will not be the empty string. The function will return FAILURE if doc or target are NULL; otherwise the function will return SUCCESS.

6.3 Important Points and Hints

1. The file document.h defines symbolic constants you need to use.
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). You can see document.c as a library.
4. You must use the constants defined in document.h (e.g. `MAX_PARAGRAPHS_LINES`). Our tests will define different values for these constants.
5. IMPORTANT: Data should only be allocated statically. You may 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.
6. If you are having trouble with your code read the debugging guide available at:

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

7. Do not modify document.h.
8. Run valgrind as you develop your code. It will help you identify memory problems.
9. 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.
10. 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.
11. You may not use strtok, strtok_r, strspn nor strcspn.
12. The next project will depend on the code you develop for this project.
13. When you remove a line you overwrite it with lines that follow. If the last line is the one being removed, there is nothing (besides adjusting the number of lines) you need to do.
14. For several load_documents calls, the data loaded will be place at the top of the document. One of the provided tests illustrates the loading that takes place in this case.
15. If we remove the only line from a paragraph, the paragraph will not disappear. The paragraph will now have 0 lines.
16. If we call replace_text(&doc, "app", "ap") on the first line below:

```
is app131.  This course will be

the result will be

is ap131.  This course will be
```

17. 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.
18. If there is nothing to compile or an executable to create when you execute "make" you will see something along the following lines: **make: Nothing to be done for 'all'.**
19. 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)
20. The number one 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) that replaces text in a line you can use it for replacing text across the whole document. Also, don't make the task harder of what it needs to be. If you are given an string and need to replace text in it, you can modify the original string, but that could 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.
21. Both init_document and reset_document can reset a document. The only difference is that one allows you to change the document's name.
22. For add_line_after you can assume paragraph_number will not be 0.
23. The replace_text function is case sensitive. For example, if looking for "cat" it will not match "Cat".
24. For this project you can assume we will not try to execute load_document more than once in a driver.

25. If you have not been using valgrind, you may not pass secret tests. 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 your code works with your own tests.
26. 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).

6.4 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/

7 Submission

7.1 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. Submit the project as usual.