CMSC 216 Project #2 Summer 2020

Document Manager **Due: Mon, Jun 22 / Wed, Jun 24, 2018, 11:30PM**

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. Notice that the next project relies on the code you will implement for this project.

There are two deadlines associated with the project. Those deadlines are:

- Mon, Jun 22, 11:30PM - Your code must pass the first two public tests (public01, public02). That is the only requirement for this deadline. We will not grade the code for style. This first part is worth .5% of your course grade (NOT .5% of this project grade). Notice you can still submit late for this part.
- Wed, Jun 24, 11:30PM - Final deadline for the project. Notice you can still submit late (as usual).

2 Objectives

To practice C structures and string manipulation.

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 Obtain the 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.

3 Specifications

3.1 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 tests by compiling the test and document.c. For example, gcc public01.c document.c.

3.2 Functions

You must implement the functions described below. Notice that for 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.

3.3 Important Points and Hints

1. You must create a file named document.c that includes the file document.h.
2. You must use the constants defined in document.h (e.g. `MAX_PARAGRAPHS_LINES`). Our tests will define different values for these constants.
3. 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.
4. If you are having trouble with your code read the debugging guide available at:

   http://www.cs.umd.edu/~nelson/classes/resources/cdebugging/
5. Do not modify document.h.
6. Run valgrind as you develop your code. It will help you identify memory problems.
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. You may not use strtok, strtok_r, strspn nor strcspn.
10. The next project will depend on the code you develop for this project.

4 Grading Criteria

Your project grade will be determined with the following weights:

Results of public tests	44%
Results of release tests	24%
Results of secret tests	18%
Code style grading	14%

4.1 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/

5 Submission

5.1 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.

5.2 Procedure

You can submit your project by executing, in your project directory (project2), the **submit** command.

6 Academic Integrity

Please see the syllabus for project rules and academic integrity information. All programming assignments in this course are to be written individually (unless explicitly indicated otherwise in a written project handout). Cooperation between students is a violation of the Code of Academic Integrity.