CMSC 216 Project 3 Fall 2021

User Interface **Due date posted on Piazza**

1 Overview

For this project, you implement a text-based user interface to your document manager in project 2. This will require adding some extra functionality to your document manager.

The objectives is to practice text parsing (analyzing) and file I/O.

1.1 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.

1.2 Grading Criteria

Your project grade will be determined as follows:

Results of public tests	30%
Results of release tests	40%
Results of secret tests	25%
Code style grading	5%

IMPORTANT restrictions:

Do not use memset, strtok, strtok_r, or memcpy. You will lose at least 30 pts if you use them.

In your scanf/fscanf/sscanf format strings, the only specifiers allowed are the following: %d %f %c %s %n or a particular word (e.g., something that is part of a command). This prevents you from using regular-expression specifiers, eg, involving `[`, `]`, `*`, `^`, `-`, `$`, `?`. You will lose at least 30 pts if you use them. See a TA if you have doubts as to what represents a regular expression.

1.3 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.

1.4 Obtaining project files

Copy the folder project3 available in the 216public projects directory to your 216 directory. *The Makefile and document.h files for this project are different from the ones in project2*. This project description can be found in the 216public project_descriptions directory.

1.5 Fixing problems with your project 2 code

After the late deadline for project2, you will be able to see results for release/secret tests in the submit server. It's in your best interest to test and fix flaws in your document manager in project 2, because the functions there will be tested again in this project. (For this, you should update your document.c in project 2, and submit your project 2 to the submit server.)

1.6 Code development and debugging

Code for parsing (analyzing strings) can get really messy. When possible, divide tasks into functions that you can test individually. Be familiar with the following material in the class page's resources: Debugging in C and Code Development.

2 Specification

2.1 Document manager update

You must add two functions to your document manager code. For this, copy document.c (but not document.h) from your project 2 into your project 3 folder.

1. `int load_file(Document *doc, const char *filename)`

   This function is similar to load_document, except that data is loaded from a file instead of from an array in memory. A new paragraph is started at the beginning of the document. An empty line indicates the beginning of a new paragraph. The function returns FAILURE if doc is NULL, filename is NULL, or opening the file fails. Otherwise the function returns SUCCESS. No error message is generated if the file cannot be opened.

2. `int save_document(Document *doc, const char *filename)`

   This function prints the document (same format as in project 2) to the specified file (overwriting the file). The function returns FAILURE if doc is NULL, filename is NULL, or the file cannot be opened. Otherwise the function returns SUCCESS. No error message is generated if the file cannot be opened.

2.2 User interface code

Your user interface code must be in a file named `user_interface.c`. It will call functions in your updated `document.c`, so compile these two files to get the executable. For example, `gcc document.c user_interface.c -o user_interface` yields the executable named `user_interface`.

A user executes your program in one of two ways:

```
user_interface
user_interface filename
```

So the program should have zero or one argument on the command line. If there is more than one argument, the program prints the following usage message to standard error, and exits with exit code `EX_USAGE`[1].

```
Usage: user_interface
Usage: user_interface <filename>
```

If there is no filename argument, the program reads its data from standard input; the program should display a prompt (represented by >) after which commands can be entered. If there is a filename argument, the program reads its data from that file and no prompt is issued.

If there is an error opening the file, your program should print (to standard error) the message ”FILENAME cannot be opened.”, where FILENAME represents the file name. The program should then exit with the exit code `EX_OSERR`.

Upon starting execution, your program initializes a single document with the name ”main_document”, and perform operations on that document as instructed by the commands (in the input data) the program reads.

Your program user_interface.c should include document.h.

2.3 File format

2.3.1 Valid Lines

An input file (or input coming from standard input) contains multiple lines with commands, and the commands are executed in the order they are encountered. A line is at most 1024 characters (including the newline character). A **valid line** takes one of three forms:

[1]All exit codes mentioned here beginning with `EX_` are available by including `<sysexits.h>`.

1. A blank line, where the line contains 1 or more spaces (as defined by the isspace() function in ctype.h).
2. A comment line, where the first non-whitespace character is a hash symbol ('#').
3. A command line, consisting of one or more strings of non-whitespace characters, where the first string is the name of a function in document.c, and any remaining strings are arguments to the function satisfying certain constraints (defined in Section 2.4 below).

For example, the following file contains only valid lines:

```
# creating a paragraph and inserting some lines
add_paragraph_after 0
add_line_after 1 0 *first line of the document

add_line_after 1 1 *second line of the document
   # let's print it
print_document
quit
```

If your program encounters an invalid line, it prints the message "Invalid Command" to standard output. Make sure you print to standard output and not to standard error. The program does not end when it reads an invalid line.

2.4 Commands

Unless output is associated with a command, the successful execution of a command does not generate any confirmation message (similar to successful execution of commands in Linux). If a command is valid but cannot be executed successfully, the message "COMMAND_NAME failed", where COMMAND_NAME represents the command, is printed to standard output (and not to standard error).

Note that a command line can be valid but cannot be executed successfully, eg, an add_paragraph_after followed by a positive paragraph number that is higher than any existing paragraph in the document.

Any number of spaces can appear between the different elements of a command, and before and after a command. A blank line (as defined above) and a comment will be ignored (no processing). When a comment or blank line is provided, and standard input is being used, a new prompt is generated.

The quit and exit commands will terminate the command processor (the executing user_interface program). The command processor will also terminate when end-of-file is seen. The commands quit or exit need not be present in the input to the processor.

1. `add_paragraph_after PARAGRAPH_NUMBER`

   This command adds a paragraph to the document. The "Invalid Command" message is generated if:

   a. PARAGRAPH_NUMBER does not represent a number
   b. PARAGRAPH_NUMBER is a negative value
   c. PARAGRAPH_NUMBER is missing
   d. Additional information is provided after the PARAGRAPH_NUMBER

   If the command is (valid but) not successfully executed, the message "add_paragraph_after failed" is generated.

2. `add_line_after PARAGRAPH_NUMBER LINE_NUMBER * LINE`

   This command adds a line after the line with the specified line number. The line to add appears after the * character. The "Invalid Command" message is generated if:

   a. PARAGRAPH_NUMBER does not represent a number

b. PARAGRAPH_NUMBER is a negative value or 0
c. PARAGRAPH_NUMBER is missing
d. LINE_NUMBER does not represent a number
e. LINE_NUMBER is a negative value
f. LINE_NUMBER is missing
g. * is missing

If the command is not successfully executed, the message "add_line_after failed" is generated.

3. `print_document`

   This command prints the document information (print_document function output). The "Invalid Command" message is generated if any data appears after print_document.

4. `quit`

   This command terminates the user interface. The "Invalid Command" message is generated if any data appears after quit.

5. `exit`

   This command terminates the user interface. The "Invalid Command" message is generated if any data appears after exit.

6. `append_line PARAGRAPH_NUMBER * LINE`

   This command appends a line to the specified paragraph. The line to add appears after the * character. The "Invalid Command" message is generated if:

   a. PARAGRAPH_NUMBER does not represent a number
   b. PARAGRAPH_NUMBER is a negative value or 0
   c. PARAGRAPH_NUMBER is missing
   d. * is missing

   If the command is not successfully executed, the message "append_line failed" is generated.

7. `remove_line PARAGRAPH_NUMBER LINE_NUMBER`

   This command removes the specified line from the paragraph. The "Invalid Command" message is generated if:

   a. PARAGRAPH_NUMBER does not represent a number
   b. PARAGRAPH_NUMBER is a negative value or 0
   c. PARAGRAPH_NUMBER is missing
   d. LINE_NUMBER does not represent a number
   e. LINE_NUMBER is a negative value or 0
   f. LINE_NUMBER is missing
   g. Any data appears after the line number

   If the command is not successfully executed, the message "remove_line failed" is generated.

8. `load_file FILENAME`

   This command loads the specified file into the current document. The "Invalid Command" message is generated if:

   a. FILENAME is missing

   b. Any data appears after FILENAME

   Assume that if FILENAME is present, it refers to a file that can be opened. If the command is not successfully executed, the message ″load_file failed″ is generated.

9. `replace_text "TARGET" "REPLACEMENT"`

   This command replaces the string ″TARGET″ with ″REPLACEMENT″. The ″Invalid Command″ message is generated if:

   a. Both ″TARGET″ and ″REPLACEMENT″ are missing
   b. Only ″TARGET″ is provided

   For this command, assume that if ″TARGET″ and ″REPLACEMENT″ are present, then there is no additional data after ″REPLACEMENT″.

   If the command is not successfully executed, the message ″replace_text failed″ is generated.

10. `highlight_text "TARGET"`

    This command highlights the string ″TARGET″. The ″Invalid Command″ message is generated if ″TARGET″ is missing.

    For this command, assume that if ″TARGET″ is present, there is no additional data after it. No fail message is associated with this command; either text is highlighted or not.

11. `remove_text "TARGET"`

    This command removes the string ″TARGET″. The ″Invalid Command″ message is generated if ″TARGET″ is missing.

    For this command, assume that if ″TARGET″ is present, there is no additional data after it. No fail message is associated with this command; either a deletion takes place or not.

12. `save_document FILENAME`

    This command saves the curent document to the specified file. The ″Invalid Command″ message is generated if:

    a. FILENAME is missing.
    b. Any data appears after the filename.

    Assume that if FILENAME is present, a file of that name can be opened. If the command is not successfully executed, the message ″save_document failed″ is generated.

13. `reset_document`

    This command resets the curent document. The ″Invalid Command″ message is generated if any data appears after reset_document. No fail message is associated with reset_document.

2.5 Important Points and Hints

1. Do not use `malloc()`, etc.
2. Do not use perror to generate error messages; use fprintf and stderr instead.
3. You can start project 3 even if you have not completed fixes to your project 2.
4. After the late deadline for project 2, you can resubmit the project to verify whether you have fixed problems in your code (this will not affect your project 2 score).
5. For this project, copy only the file document.c from project 2. Do not copy the file document.h or Makefile from project 2.

6. Add the load_file and save_document functions (and any functions that support them) to document.c (not user_interface.c). If you have support functions, add the prototypes at the top of document.c. Do not modify the document.h file of project 3.
7. Create a file named user_interface.c and implement the user interface in this file (and not in document.c). The user_interface.c file will have a main function that reads commands and process them (using the functions in document.c). If you introduce support functions in user_interface.c, add their prototypes at the top of user_interface.c.
8. Do not include .c files using #include. That means you may not include document.c in your user_interface.c file. You will include document.h.
9. You can use header files in user_interface.c that provide support as long as it does not violate the rules we defined in the project. You should be able to implement user_interface.c with the following header files:

```
#include <stdio.h>
#include <string.h>
#include <sysexits.h>
#include <stdlib.h>
#include <ctype.h>
#include "document.h"
```

10. The streams_example.c lecture example can help you during the implementation of this project. Feel free to use any lecture/lab code.
11. Assume a filename does not exceed 80 characters.
12. If you remove a line, that line should not be printed (no blank line for it).
13. In add_line_after command, everything after * (including spaces) is part of the line. For example:

```
> add_paragraph_after 0
> add_line_after 1 0 *first line of the document
> add_line_after 1 1 *      more data here
> print_document
Document name: "main_document"
Number of Paragraphs: 1
first line of the document
      more data here
> quit
```

14. Defining the size of any string variable to be 1024 is fine. Actually, you can even have a smaller size as we don't plan to add lines than exceed 100 characters.
15. Assume that when a string appears, it will be enclosed in double quotes. Don't worry about cases like the following:

```
highlight_text "every
replace_text "CS   "Computer Science"
replace_text "CS"   Computer Science"
replace_text CS"   "Computer Science"
```

16. All commands (e.g., quit", "exit", "add_paragraph", etc.) will be in lowercase.
17. Regarding replace_text:
    a. Assume "TARGET" is never the empty string (`""`).
       So we will never ask you to execute `replace_text "" ""`.
    b. "REPLACEMENT" can be the empty string (e.g., `replace_text "of" ""`).

18. Although you should always avoid code duplication, for this project, if you are in doubt about code duplication in the code you have written it, ignore it. We will not be penalizing for it in this project. The nature of this project makes it hard to avoid some level of duplication, which is difficult to factor out in a function.
19. If you remove all the lines from a paragraph, the paragraph is not removed (paragraph count stays the same).
20. The atoi function returns 0 if the provided string does not represent an integer. Keep this in mind if you use this function.
21. IMPORTANT: Remember that breaking down the desired computation into functions allows you to control the complexity of the code you need to implement. Think of functions you can implement that will simplify the implementation and testing process.
22. We have seen how abusing the use of the continue statement may lead to code that is difficult to understand. Be careful if you decide to use it. We recommend you don't use it at all.
23. You should use valgrind for this project as follows: `valgrind user_interface public01.in`

    The following is incorrect: `valgrind public01`
24. To understand why multiple prompts (eg, >>>>>>) arise in the public test outputs when using input redirection, see the information at:

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

    under the section "Program Output When Using Input/Output Redirection".

2.6 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).
- Do not use global variables.
- Feel free to use helper functions for this project; just make sure to define them as static.
- Follow the C style guidelines available at:

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

3 Submission

To submit your project, execute the **submit** command in your project directory,