CMSC 216 Project #3 Summer 2022

User Interface **Due: Fri, Jun 24 / Tue, Jun 28, 2022, 11:55 pm**

1 Overview

1.1 user_interface program

For this project you will write a text-based user interface to the document manager system you implemented in project #2. In addition, you will add some extra functionality to your system. There are two deadlines associated with the project. Those deadlines are:

- Fri, Jun 24, 11:55 pm - 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). You can still submit late for this part.
- Tue, Jun 28, 11:55 pm - Final deadline for the project. Notice you can still submit late (as usual).

2 Objectives

To practice text parsing (analyzing) and file I/O.

3 Grading Criteria

Your project grade will be determined with the following weights:

Results of public tests	20%
Results of release tests	45%
Results of secret tests	30%
Code style grading	5%

3.1 Obtain the project files

To obtain the project files copy the folder project3 available in the 216 public directory to your 216 directory. Keep in mind that the Makefile and document.h files for this project are different from the ones used in project2.

3.2 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. A TA during office hours (and only during office hours) will be able to show you any test and why the test failed (if that is the case). You are responsible for fixing your code before submitting this project. Keep in mind that if you passed all the project2 tests that does not mean you don't have bugs. In this project we will be testing your document functions again so it is in your best interest to test your code thoroughly.

4 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.

5 Announcements

1. The approach you should be using to develop projects is as follows:
   a. Write a little bit of code (incremental code development).
   b. Test your code.
   c. Even if your code is generating the correct results, check your code using valgrind, splint, and the -fsanitize=address gcc option. This will improve the probability of passing secret tests.
   d. Backup your code by submitting often.
   e. Submitting often and checking results in the submit server is important as sometimes code works in grace, but not in the submit server (what one student called in Piazza "Falling from Grace" :))
   f. Go back to the first step.
2. Be familiar with the following information:
   a. Debugging in C: http://www.cs.umd.edu/~nelson/classes/resources/cdebugging/
   b. Code Development: http://www.cs.umd.edu/~nelson/classes/resources/cdebugging/development_strategy/
3. In this project you will be parsing (analyzing strings). Code can get really messy if you do not split tasks as needed. When possible, divide tasks into functions that you can test individually.

6 Specification

6.1 Document manager update

You need to add two functions to your document manager system. Remember to use the provided document.h file (not the one from project #2).

1. `int load_file(Document *doc, const char *filename)` - This function is similar to load_document, except data will will be loaded from a file instead of using an array. By default a paragraph will be added and any blank lines (line with only spaces as defined by isspace()) will mark the beginning of a new paragraph. The function will fail and return FAILURE if doc is NULL, filename is NULL, or if opening the file failed; otherwise the function will return SUCCESS. No error message will be generated if the file cannot be opened.
2. `int save_document(Document *doc, const char *filename)` - This function will print the paragraphs associated with a document to the specified file (overwriting the file). Each paragraph will be separated by a newline. The function will fail and return FAILURE if doc is NULL, filename is NULL, or the file cannot be opened; otherwise the function will return SUCCESS. No error message will be generated if the file cannot be opened.

6.2 Method of operation

Your program will be in a file named `user_interface.c`. A user calls your program in one of two ways (assuming the executable is named user_interface):

```
user_interface
user_interface filename
```

The program should have zero or one arguments (in addition to the executable name) on the command line; if there are more the program prints the following usage message to standard error, and exits with exit code `EX_USAGE` [1].

[1] This and the other exit codes beginning with `EX_` mentioned here are all obtained by including `<sysexits.h>` in your C program file.

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

If there is no file specified when the program is started, the program should read its data from standard input. The program will display a prompt (represented by >) after which commands will be entered. If a file is named, however, the program reads its data from that file; in this case no prompt will be used.

In case of 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 will then exit with the exit code `EX_OSERR`.

Upon starting execution your program should initialize a single document with the name "main_document", and perform operations on that document as instructed by the commands the program reads.

Make sure you name the file with your program user_interface.c. This program will include document.h (the version provided for this project and not the one from project #2).

6.3 File format

6.3.1 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. No valid line can be more than 1024 characters (including the newline character). A valid line takes one of three forms:

1. A comment, where the first non-whitespace character is a hash symbol ('#').
2. A command, where the line is composed of one or more strings of non-whitespace characters.
3. A blank line, where the line contains 1 or more spaces (as defined by the isspace() function in ctype.h).

For example, the following file contains 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
```

Valid commands must follow one of the formats specified in Section 6.4 below.

6.3.2 Invalid Lines/Commands

If your program encounters an invalid line it should print the message "Invalid Command" to the standard output. Make sure you print to the standard output and not to the standard error. An invalid line includes not only an invalid command, but a command without the expected values. For example, the add_paragraph_after command requires an integer. If the value provided is not an integer, the command will be considered invalid. The program will not end when an invalid command is provided.

6.4 Commands

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

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 will be generated.

The quit and exit commands will end/terminate the command processor. The command processor will also terminate when end of file is seen. The commands quit or exit need not be present in a file.

1. `add_paragraph_after PARAGRAPH_NUMBER`

   This command will add a paragraph to the document. The ”Invalid Command” message will be generated when:

   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 cannot be successfully executed the message ”add_paragraph_after failed” will be generated.

2. `add_line_after PARAGRAPH_NUMBER LINE_NUMBER * LINE`

   This command will add a line after the line with the specified line number. The line to add will appear after the * character. The ”Invalid Command” message will be generated when:

   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 cannot be successfully executed the message ”add_line_after failed” will be generated.

3. `print_document`

   This command will print the document information (print_document function output). The ”Invalid Command” message will be generated if any data appears after print_document.

4. `quit`

   This command will exit the user interface. The ”Invalid Command” message will be generated when any data appears after quit.

5. `exit`

   This command will exit the user interface. The ”Invalid Command” message will be generated when any data appears after exit.

6. `append_line PARAGRAPH_NUMBER * LINE`

   This command will append a line to the specified paragraph. The line to add will appear after the * character. The ”Invalid Command” message will be generated when:

   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 cannot be successfully executed the message ”append_line failed” will be generated.

7. `remove_line PARAGRAPH_NUMBER LINE_NUMBER`

   This command will remove the specified line from the paragraph. The ”Invalid Command” message will be generated when:

   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 cannot be successfully executed the message ”remove_line failed” will be generated.

8. `load_file FILENAME`

   This command will load the specified file into the current document. The ”Invalid Command” message will be generated when:

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

   If the command cannot be successfully executed the message ”load_file failed” will be generated.

9. `replace_text "TARGET" "REPLACEMENT"`

   This command will replace the string ”TARGET” with ”REPLACEMENT”. The ”Invalid Command” message will be generated when:

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

   For this command you can assume that if ”TARGET” and ”REPLACEMENT” are present there is no additional data after ”REPLACEMENT”.

   If the command cannot be successfully executed the message ”replace_text failed” will be generated.

10. `highlight_text "TARGET"`

    This command will highlight the string ”TARGET”. The ”Invalid Command” message will be generated when ”TARGET” is missing.

    For this command you can assume that if ”TARGET” is present there is no additional data after it. Notice no fail message is associated with this command; either the text was highlighted or not.

11. `remove_text "TARGET"`

    This command will remove the string ”TARGET”. The ”Invalid Command” message will be generated when ”TARGET” is missing.

    For this command you can assume that if ”TARGET” is present there is no additional data after it. Notice no fail message is associated with this command; either a deletion took place or not.

12. `save_document FILENAME`

    This command will save the curent document to the specified file. The ”Invalid Command” message will be generated when:

a. FILENAME is missing.

b. Any data appears after the filename.

If the command cannot be successfully executed the message "save_document failed" will be generated.

13. `reset_document`

    This command will reset the curent document. The "Invalid Command" message will be generated when any data appears after reset_document. Notice no fail message will be associated with reset_document.

6.5 Important Points and Hints

1. Data should only be allocated statically. You may not use `malloc()`, etc.
2. Do not use perror to generate error messages; use fprintf and stderr instead.
3. IMPORTANT: You may not use strtok or strtok_r in this project. You will be penalize at least 30 pts if you use them.
4. IMPORTANT: A regular expression is a combination of characters that represents a set of strings. The format string of a scanf/fscanf/sscanf can have regular expressions and the scanf can then recognize any string in the set defined by the regular expression. Using regular expressions will not allow you to practice string parsing (analysis). For this project your format strings should only have the following: %d %f %c %s %n or a particular word (e.g., something that is part of a command). If you are using anything else, for example:

   ```
   [, ], *, ^, - $, ?
   ```

   you are relying on regular expressions and you will be penalized at least 30 pts. See a TA if you have doubts as to what represents a regular expression.
5. You need to fix any problems associated with your project #2, but you do not need to wait until you fix them to start project #3.
6. After the late deadline for project #2, you can resubmit the project (this will not affect your score). You may want to resubmit to verify you have fixed any problems your code has.
7. For this project only copy the file document.c from project #2. Do not copy the file document.h; you need to use the one we provided for project #3. Also use the Makefile we have provided for project #3.
8. Add the load_file and save_document functions (and any functions that support them) to document.c (not user_interface.c). You can have support functions and you don't need to add the support functions prototypes to document.h. Just add the prototype(s) at the top of document.c. Do not modify the document.h file we provided for project #3.
9. 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). You can add support functions to the user_interface.c file; just add the prototype at the top of user_interface.c.
10. 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.
11. 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. For this 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"
```

12. The streams_example.c lecture example can help you during the implementation of this project. Feel free to use any lecture/lab code.
13. You can assume a filename will not exceed 80 characters.
14. If you remove a line, that line should not be printed (no blank line for it).
15. There can be no spaces between the # and the comment (e.g., #this is a comment).
16. Regarding add_line_after: 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
```

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

```
highlight_text "every

replace_text "CS   "Computer Science"

replace_text "CS"   Computer Science"

replace_text CS"   "Computer Science"
```

19. All commands (e.g., quit", "exit", "add_paragraph", etc.) will be in lowercase.
20. Regarding empty string (""") in replace_text:
    a. You can assume we will never try to execute replace_text "" ""
    b. You can assume "TARGET" will never be the empty string ("")
    c. "REPLACEMENT" can be the empty string (e.g., replace_text "of" "")
21. 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 that can be difficult to factor out in a function.

22. If you remove all the lines from a paragraph, the paragraph will not be removed (paragraph count will stay the same).
23. The atoi function returns 0 when the provided string does not represent an integer. Keep this in mind if you use this function.
24. IMPORTANT: Remember that breaking down the computation you need to implement 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.
25. 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 and we recommend you don't use it at all.
26. You should use valgrind for this project as follows:

    valgrind user_interface public01.in

    The following is incorrect:

    valgrind public01
27. To understand the multiple `(e.g., >>>>>>)` in the public tests output, when using input/output redirection check, see information available at:

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

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

6.6 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/

7 Submission

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