COMP1921 16s2 Assignment

Text Editor

Aims

This exercise aims to give you practice in dealing with dynamic data structures, specifically linked lists. The goal is to complete the implementation of a medium-sized program that can do simple text editing.

Assessment

Marks:	15 (Criteria: correctness, readability, design)
What to submit:	`lite.c`, `lite.h` (NOTE: You should submit `lite.h` even if you decide to leave it empty.)
Submit before:	Midnight, Friday, Week 13 (28 October)

Change Log

We may make minor changes to the spec to address/clarify some outstanding issues. These may require minimal changes in your design/code, if at all. Students are strongly encouraged to check the change log regularly.

Version 1: Released on 18 September 2016.

Background

The assignment is to write a simple line-based text editor called ***lite*** (which stands for **li**ne **t**ext **e**ditor). The editor **lite** stores lines of text in a linked list: each line of text is stored in a node of the linked list. If there are *n* lines of text, they are numbered from 1 to *n* for the user. At every point in time the editor has a **current line number** (which is an index between 1 and *n*). When **lite** is started up, the current line number is 1. The line-based commands allow the user to print lines, to delete the current line, to insert and append lines before or after the current line and to go-to a particular line. There are also file-oriented commands that enable the user to save changes and quit the editor, to force an exit or to change the name of the file. Note that lite does not contain a search command or contain any way of changing text apart from deleting 'old' lines and appending and inserting new lines.

The main objectives of this assignment are:

- use and manipulate dynamic data structures (like linked lists) to solve a complex problem
- learn how to implement a linked list data structure and functions for maintaining the data structure
- gain experience with implementing a more comprehensive functionality for editing text files
- write a properly documented C program that adheres to the course Style Guide

NOTE: You should think carefully about the appropriate data structures and algorithms to use in your program. Before starting to write any code it is important that you fully understand the problem and determine the data structures that you will require and the algorithms to use. It is highly recommended that you start coding only after you have spent some time on these

considerations. In particular, you must not make any assumptions about the number of lines in a text file; this means that you must use dynamically-allocated linked-list data structure to manage a text file.

What the editor should do

A file name is the only optional argument to the **lite** command. So you execute **lite** either by:

```
prompt$ ./lite
```

or

```
prompt$ ./lite filename
```

where *filename* is a file name that may or may not exist. If there are more arguments, then the usage error message from **lite** is:

```
Usage: ./lite [filename]
```

The square brackets indicate that *filename* is optional.

Examples of Use

When **lite** is waiting for user input it uses the prompt '?'. When text is printed, each line is preceded by its line number. The current line number is indicated by an arrow.

Example 1

The user can start up the editor without a file name, print the help command and then quit.

```
prompt$ ./lite
? h
Commands are (in upper or lower case):
        q:      quit
        s:      save
        x:      force exit
        f <filename>: the file is called <filename>
        h:      print this help message
        d:      delete current line
        a:      append after current line, terminated by '.'
        i:      insert before current line, terminated by '.'
        p:      print all lines
        .:      print current line
        +:      increment line and print
        <return>: same as '+'
        -:      decrement line and print
        number: make 'number' the current line
? q
bye
prompt$
```

Example 2

In the next example, if the file Mackellar.txt contains the text

```
I love a sunburnt country,
A land of sweeping plains,
```

```
Of ragged mountain ranges,
Of droughts and flooding rains.
I love her far horizons,
I love her jewel-sea,
Her beauty and her terror -
The wide brown land for me!
```

then an example of an edit session that involves the user

a. opening the file Mackellar.txt,
b. printing its contents by entering the command *p*,
c. moving to the 8th line in the file (this is the last line) by entering *8*,
d. appending 4 lines (consisting of a blank line, the name of the poem, the author and her year of birth and death)
    - notice the text is terminated by a line containing the symbol '.' only (just like UNIX's **ed**)
    - the current line number after an insertion is the last line to be entered
e. entering an incorrect command *v*,
f. printing the contents again by entering *p*,
g. saving the text by entering *s*, and finally
h. quitting the editor by entering *q*

is shown below:

```
prompt$ ./lite Mackellar.txt
Existing file "Mackellar.txt"
? p
--->   1: I love a sunburnt country,
Line   2: A land of sweeping plains,
Line   3: Of ragged mountain ranges,
Line   4: Of droughts and flooding rains.
Line   5: I love her far horizons,
Line   6: I love her jewel-sea,
Line   7: Her beauty and her terror -
Line   8: The wide brown land for me!
? 8
--->   8: The wide brown land for me!
? a

"My Country"
Poem by Dorothea Mackellar
(1885--1968)
.
? v
Unknown command: ignoring
? p
Line   1: I love a sunburnt country,
Line   2: A land of sweeping plains,
Line   3: Of ragged mountain ranges,
Line   4: Of droughts and flooding rains.
Line   5: I love her far horizons,
Line   6: I love her jewel-sea,
Line   7: Her beauty and her terror -
Line   8: The wide brown land for me!
Line   9:
Line  10: "My Country"
Line  11: Poem by Dorothea Mackellar
--->  12: (1885--1968)
? s
Saving file "Mackellar.txt"
? q
bye
prompt$
```

Example 3

In the next example, the user steps down an existing C program to a particular line, deletes the line and inserts a new line.

```
prompt$ ./lite hw.c
Existing file "hw.c"
? p
--->   1: #include <stdio.h>
Line   2: #define NUMBER 10
Line   3: int main(void) {
Line   4:    int i;
Line   5:    for (i=0; i<NUMBER; i++) {
Line   6:       printf("hello, world!\n");
Line   7:    }
Line   8:    return 0;
Line   9: }
? +
--->   2: #define NUMBER 10
? +
--->   3: int main(void) {
? +
--->   4:    int i;
? +
--->   5:    for (i=0; i<NUMBER; i++) {
? +
--->   6:       printf("hello, world!\n");
? d
--->   6:    }
? i
      printf("goodbye, world!\n");
.
? p
Line   1: #include <stdio.h>
Line   2: #define NUMBER 10
Line   3: int main(void) {
Line   4:    int i;
Line   5:    for (i=0; i<NUMBER; i++) {
--->   6:       printf("goodbye, world!\n");
Line   7:    }
Line   8:    return 0;
Line   9: }
? s
Saving file "hw.c"
? q
bye
prompt$
```

Note that it would have been faster, of course, to have typed in the line number, 6, rather than step down line by line.

Example 4

In the final example, the user starts the editor up, does a print to check that there is no text, inserts a poem from Banjo Paterson, does another print, sets the file name to Banjo.txt, saves the text and quits.

```
prompt$ ./lite
? p
Empty file
? i
A land of sombre, silent hills, where mountain cattle go
By twisted tracks, on sidelings steep, where giant gumtrees grow
And the wind replies, in the river oaks, to the song of the stream below.
.
```

```
? p
Line   1: A land of sombre, silent hills, where mountain cattle go
Line   2: By twisted tracks, on sidelings steep, where giant gumtrees grow
--->   3: And the wind replies, in the river oaks, to the song of the stream below.
? f Banjo.txt
Creating file "Banjo.txt"
? s
Saving file "Banjo.txt"
? q
bye
prompt$
```

Example 5

Note that a quit is not possible if the user has changed the file in any way. For example:

```
prompt$ ./lite
? i
1
2
3
.
? s
No filename. Use '-f <name>' command
? q
Cannot quit as file has changed. Use 'x' to force exit
? f FRED.txt
Creating file "FRED.txt"
? s
Saving file "FRED.txt"
? q
bye
prompt$
```

In this example, the user could have instead used an 'x' command to leave the editor without saving.

Example 6

You cannot step beyond the last line or the first line in the buffer:

```
prompt$ ./lite Mackellar.txt
Existing file "Mackellar.txt"
? p
--->   1: I love a sunburnt country,
Line   2: A land of sweeping plains,
Line   3: Of ragged mountain ranges,
Line   4: Of droughts and flooding rains.
Line   5: I love her far horizons,
Line   6: I love her jewel-sea,
Line   7: Her beauty and her terror -
Line   8: The wide brown land for me!
Line   9:
Line  10: "My Country"
Line  11: Poem by Dorothea Mackellar
Line  12: (1885--1968)
? 123
Line number does not exist: command ignored
? -
--->   1:I love a sunburnt country,
? -
--->   1:I love a sunburnt country,
? -
--->   1:I love a sunburnt country,
```

```
? 12
--->  12:(1885--1968)
?
--->  12:(1885--1968)
? +
--->  12:(1885--1968)
? +
--->  12:(1885--1968)
? +
--->  12:(1885--1968)
? q
bye
prompt$
```

Possible command sequences

There are many other edit sequences possible. For example, consider:

- no filename
  - `h, q`
  - `h, x`
  - `i, q` (error)
  - `i, s` (error)
  - `i, f` (error)
  - `i, f <..>, q` (error)
  - `i, f <..>, s, q`
  - `i, i, i, i, i, f <..>, s, q`
  - all of the above with `a` instead of `i`
  - `i, i, i, d, d, d, f <..>, s, q`
  - `i, d, a, d, f <..>, s, q`
  - combination of `i` and `d`, end with `x`
  - illegal letters such as `w`, `e`, `r` etc
- with a filename (say 20 lines)
  - filename doesn't exist (error)
  - `+, +, +, ..., -, -, -, ...`
    - can you step beyond the last line?
    - can you step back to before the first line?
  - many <return>, `-, -, -, ...`
  - combination of `+` and `i`
  - <middleline>, `i`
  - `1, i`
  - <lastline>, `a`
  - <nonexistentline> (error)
  - `d` all lines, `x`
  - `d` all lines, `s`, `q`

  In the case of '`x`', no changes to the file should be made.

Approach

Do not try to write the whole program at once. Break the development into pieces of functionality and implement the easiest first:

Stage 1 – Reading Text

1. Write a program that reads lines of text from stdin and creates a linked list in which each node contains a single line.
    - You will need a function that prints all the nodes in a linked list to check that the read was successful.

    You are allowed to copy chunks of code from the `list.c` library from lectures, if you find it helpful. If you use code like this, you *must* mark it with a comment to indicate its source, e.g.

    ```
    // This code adapted from COMP1921 lecture example
    ```

2. Change the program to read from a file, where the name of the file is given on the command line.

Stage 2 – Navigating

3. Change the program to accept simple commands from the keyboard:
    - *p* to print the complete contents of the linked list (it calls the function you wrote earlier)
    - *h* to print the help menu
    - *x* to exit the program (you will need to write a function to free the linked-list nodes)
    - if the command is not one of the above, an appropriate message should be printed and the program should wait for the next command

4. Now add the 'move around' commands to the program:
    - an integer to go to a particular node in the linked list and print the corresponding line
        - this raises the issue of the *current line* number, which is central to the workings of the editor
    - + and - to print the line before and after the *current line*
        - it should be possible to enter any number of + and - to step through the linked list
        - address the problem of what happens to the *current line* when you reach the start and end of the linked list
    - a <return> (i.e., `'\n'`) is equivalent to a +, so that needs to be implemented as well

Stage 3 – Editing

5. Now add the delete and insert/append operations:
    - do *d* first; remember to free the linked-list node when the line of text is deleted
    - be mindful of what happens to the current line when you have inserted/appended or deleted lines

6. Add file I/O to the program, which involve the *s*, *q* and *f* commands:
    - the *f* command is tricky as it requires a filename argument
    - the *s* command will write the complete linked list to a file
    - the *q* command requires the program to know whether the lines of text have changed or not

Hints

You may use fixed-length arrays to read a line of text from stdin:

```
#define LINELENGTH  1000  // max length of a line
```

and to read a filename or an editor command:

```
#define FNAMELENGTH 100   // max length of a file name
#define CMDLENGTH   102   // max length of a command (e.g.'f' + ' ' + filename)
```

However, you should use the heap to store the lines in the linked list. The nodes in the linked list, each storing one line of text, may be quite simple:

```
typedef struct _node {
   char *data;          // a line of text
   struct _node *next;
} Line;
```

but you may place more data in this data structure if you wish.

UNIX editor "ed"

The UNIX system has its own simple (but powerful) text editor, called **ed**, which uses similar syntax to **lite**. For example, **ed** uses the *p*, *d*, *a*, *i* commands with a dot as terminator, but **ed** does not have a prompt. You may wish to play with **ed** to get a feeling for how the editor should work.

For a more detailed understanding of what's required, a compiled solution to the assignment (i.e. a working `lite` program) has been provided for you in

```
~cs1921/bin/lite
```

Your program should produce *exactly* the same output as this executable.

You can test your program using

```
prompt$ ~cs1921/bin/dryrun assignment
```

Note: *Passing the dryrun test does not guarantee that your program is correct. You should thoroughly test your program with your own test cases.*

Submission

The due date is Friday, Week 13 (28 October), at 23:59:59.

15% penalty will be applied to the (maximum) mark for every 24 hours late after the deadline. No submissions will be accepted after 4 November 2015.

The submit command (**available in week 10**) is:

```
give cs1921 assignment lite.c lite.h
```

The file `lite.h` may be empty, but should be present.

- This file is there for your convenience: you may also place all definitions in your main program if you wish.

Be sure that your program compiles using the usual compiler flags (on a computer in CSE).

You can submit as many times as you like — later submissions will overwrite earlier ones. You can check that your submission has been received by using the following command:

```
1921 classrun -check
```

Additional information may be found in the FAQ and will be considered as part of the specification for the project. Questions relating to the project can also be posted to the Assignment web page. If you have a question that has not already been answered on the FAQ or on Assignment webpage, you can email it to your tutor.

Plagiarism Policy

Group submissions will not be allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise (including submissions for similar projects in previous years, if applicable) and serious penalties will be applied, particularly in the case of repeat offences.

DO NOT COPY FROM OTHERS; DO NOT ALLOW ANYONE TO SEE YOUR CODE

Please refer to the on-line sources to help you understand what plagiarism is and how it is dealt with at UNSW:

- Academic Integrity and Plagiarism

Marking

This project will be marked on functionality in the first instance, so it is very important that the output of your program be EXACTLY correct (identical to the output of the sample solution). Since your submssion will be automarked, if it does not exactly match the expected output, it will be recorded as incorrect. it is *your* responsibility to test it appropriately to ensure that it does produce recognisably correct output.

Submissions which score correct on the automarking will be awarded full marks on the testing component of the marking. Submissions which score very low on the automarking will be looked at by a human and will receive some marks, provided the code is well-structured and commented.

Programs that generate compilation errors will receive a very low mark, no matter what other virtues they may have. In general, a program that attempts a substantial part of the job and does that part correctly will receive more marks than one attempting to do the entire job but with many errors.

The marking breakdown is as follows:

- Stage 1 — 4 marks
- Stage 2 — 4 marks
- Stage 3 — 4 marks
- Comments, programming style, clarity and adherence to the style guide — 3 marks.

Finally ...

Have fun!