CSE 3150 — Essentials of C++ — Yufeng Wu — Spring 2020
Project Milestone 2: Adding More Features to the Text Editor. Due: 04/22/20.

By now you should have implemented the basic functionalities for the text editor. We need to continue to implement more essential features of the editor in the milestone two. In this milestone, you will make your editor behave close to a working editor, which you can actually use to do some real text editing. For this, you will need to allow larger document, one that may contain more texts than those can be fitted into the text view. You will also need to support basic file I/O.
Reminder: you should try to practice object orientation in this milestone. As I said many times, applying OO in a real project settings is the best way to learn this important design paradigm.

There are the main objectives for the second milestone of project:

1. Basic file I/O: read and write to a file.
2. Formatting of texts for proper display of large documents in the view.

1 Basic file I/O

Your text editor should now be able to read from a file or write to a file. Here are the main usages:

1. Your editor should now always be run with a file name. For example, you should launch the editor like:
   ./myeditor myfile.txt
   That is, your editor will know what files to read and write when the editor is run. This avoids the need for prompting the user for file names.
2. If there is an existing file with the entered file name, read from that file and populate the stored text in your text document.
3. If there is no file with the entered file name (i.e., the file is new), your document should be initialized to be empty.
4. In either case, upon exit of your document, your editor will always write to the file with the input filename what is in the text (even if the text document is empty).

Caution: Note that the editor doesn't confirm with the user whether the editor is to write to an existing file. Thus, be careful when running your editor: don't let the editor accidentally destroy a file you want to keep.

2 Formatting of texts for view display

In the description of the milestone one, I said you don't need to consider larger documents. To make your text editor practical, you now need to implement new features to allow your editor to handle larger text documents (those cannot fit to a single view).

There are various ways to support this. In this project, I have decided to let you support document with **row-paragraph** structure. That is, a document consists of a number of paragraphs; each paragraph consists one or multiple rows. Each paragraph corresponds to those text that are entered by the user without pressing enter. That is, if the user enters the text without pressing "enter", these texts are considered to be in the same paragraph; a enter would create a new paragraph: if "enter" is pressed at the end of a paragraph, the new paragraph created would be empty; otherwise if "enter" is pressed in the middle of a paragraph, it breaks the paragraph into two at the position where enter is pressed.

Line breaking When the paragraph gets longer than what can be fitted into a single row, the editor will need to perform line breaking. That is, the paragraph needs to be displayed as more than one rows (called line breaking). There are various ways for line breaking. In this assignment, you need to implement the following line breaking: don't break up the word; move the whole word to a new line when needed. For example, if there is a word "Mississippi" and a row can only have space to display "Mississi". Then move the entire word "Mississippi" to the next row. You may assume there is no single word that is longer than the entire row.

To display a large document which cannot be fitted to a single view, you should support **pages**. A page consists rows that can be fitted into the text view. When it comes to display a document, we may think of a document as consisting of pages, each page consisting a number of rows. The user can navigate through pages using "Up" and "Down" arrow keys. In particular, at the end of the current page, if a user presses "Down", the editor would display the next page (if exists). Note: the *whole* next page would be displayed; the cursor would be placed at the first row. Similarly, if the user presses "Up" at the first row of the current page, the editor would display the whole previous page; and the cursor would be placed at the last row of the page.

The following are more specific user actions to support.

1. Cursor movement. The user can move cursor within the paragraph or between paragraphs: if a user presses "Left" at the first character of a row, the cursor should be placed to the end of previous row (if exists); if a user presses "Right" at the last character of a row, the cursor should be placed at the beginning of the next row (if exists). If a user presses "Down" at the last row of the page, move to the next page (which will replace the current page in the view) if exists; if a user presses "Up" at the first row of the page, move to the previous page (which will replace the page in the view) if exists.

2. Text insertion. The user can type at any position of the text to insert new texts into the current paragraph. You must support line breaking (as described above) when needed.

3. "Enter". If you press "enter" at the end of a row, you will create a new (empty) paragraph after the current paragraph. If you press "enter" within a paragraph, this breaks the current paragraph into two paragraphs. You must properly format both paragraphs.

4. Text removal. Use backspace to remove the text in a paragraph. You must ensure the paragraph remains in proper formatting. That is, text removal may cause formatting changes in line breaking. If you press backspace at the first column (position 0) of a paragraph and there is some paragraph before it, then you will merge the two paragraphs. Again, the paragraph needs to follow proper line breaking format.

5. Undo/redo: as before, all the editing actions should be undoable/redoable. Here, ctrl-z for undo and ctrl-y is for redo.

3 Design and implementation

I strongly suggest you to apply OO principles and design patterns in your code. Working on a real project is the best way to learn how to do OO. Try to make your code flexible and accommodate for changes. In particular, I recommend you to apply the following design patterns:

1. **Strategy**: use "strategy" pattern to encapsulate line breaking algorithm into an object. This can make the design of your code cleaner and easier to manage.

2. **Composite**: use "composite" pattern to implement the document structure: you have a composition (document), paragraph and rows. This is where you can apply the composite pattern.

4 What to submit?

At the end of this assignment, each student must submit the following:

1. A short report on the key features you have implemented. We also like to see how you implement the text editor: how did you apply object orientation in your project? did you use some *design patterns*? if so, what are they and how you applied? You must also provide a simple user manual, which teaches the user how to use your text editor. This should be submitted a single PDF file. Provide the features in tabular format as shown below:

Feature	Status	Design pattern	User Manual
Feature name	functional/ bugs/ not implemented	Design pattern Used	User manual for feature with description for users for using the feature.
e.g.			
Text Entry	Implemented	Observer/...	Keyboard entry for all letters are supported.
Undo Text	Bugs		Press Ctrl+z Known Issue: Unable to undo text if the ¡return¿ key is entered.

2. Source code. You must include a Makefile. Your code should be able to build by typing "make" at the main source code folder. You will lose a significant portion of grade if your code cannot compile this way. Note: we may or may not look at your code for grading.