Middle East Technical University | Department of Computer Engineering

CENG 213

Data Structures

Fall 2015-2016

Homework 1 - Text Editor Implementation with C++

Due date: 23 November 2015, Monday, 23:55

1 Objectives

This assignment aims to get you familiar with data structure concepts by implementing a basic text editor using C++ classes and templates.

Keywords: Doubly Linked List, Stack, Gap Buffer, Classes and Templates in C++

2 Problem Definition

You are working at *BTC* as Software Engineer and you are newly assigned to an ongoing project called *Fancy Text Editor*. In this project, some parts have already been completed and your responsibility is to finish remaining parts.

As a short summary of the project, Fancy Text Editor is an editor in which lines are stored as *Doubly Linked List* and each line is made of a data structure called *Gap Buffer*. The editor is capable of inserting and deleting characters, undo previous actions, moving cursor up, down, left and right. Moreover, the project has a graphical part and it is responsible for gathering user input and showing current state of the editor to the user. The following figure shows the big picture of the project:

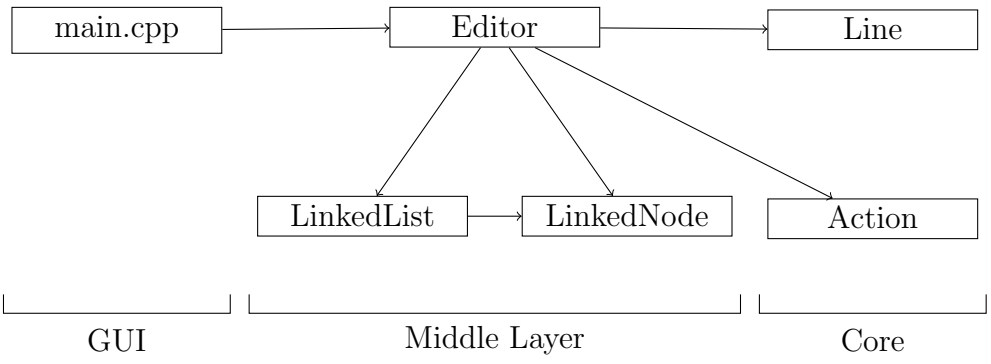

Figure 1: Dependency graph of the classes/files.

As this project was already started when you joined the company, the most difficult parts such as the GUI and the core are already implemented. Therefore, you are expected to implement only the middle layer.

The middle layer consists of *Editor*, *LinkedList* and *LinkedNode* classes. Details of these classes are given in the following section.

As an internal policy of the company, your codes will be tested by automatically generated test cases which are not visible to you. Based on the results of these tests, you will get your overall performance score (and perhaps a little bonus to your salary).

3 Project Details

3.1 Core

Core part consists of *Line* class, *Action* class and *ActionType* enumerator.

3.1.1 Line

This class implements a data structure called as *Gap Buffer*. Since this part is already implemented, you do not need to know the details of the *Gap Buffer*. However, you should understand the public functions of *Line* class because *Editor* class uses them. You can read the comments of the public functions in *Line.h* to learn more about them.

3.1.2 Action

Action class is used to store an action that user has made. Undo operation in *Editor* class will be implemented using this class. Similar to *Line* class, this class is already implemented and you do not need to know details of this class.

3.1.3 ActionType

ActionType enumerator is defined in *Action.h* file and used to identify the type of the action that user made. Action type can be $INSERT\_CHAR$, DEL, $BACKSPACE$ and $INSERT\_LINE$. You will use these types while creating *Action* class to store the history of the user actions.

3.2 Middle Layer

Your main responsibility is to implement public functions of the classes in this layer. Each function is provided with comments on top of it explaining what this function is for and what the exceptions from this function are. Thus, details of the functions can be found in the corresponding header file.

3.2.1 LinkedNode

This template class is used as a container node for linked list. It has three private member variables which are pointers to the next and previous *LinkedNode*s and actual data of the node. You will implement *getters* and *setters* of these private member variables in this class.

```
template<class T>
class LinkedNode {
private:
  LinkedNode<T> *prev;
```

```
  LinkedNode<T> *next;
  T data;
public:
  LinkedNode();

  LinkedNode<T>* getNext() const;
  LinkedNode<T>* getPrev() const;
  T* getData();

  void setNext(LinkedNode<T> *newNext);
  void setPrev(LinkedNode<T> *newPrev);
  void setData(T& data);
};
```

Listing 1: Definition of LinkedNode template class

3.2.2 LinkedList

This template class is a C++ implementation of data structure, *Doubly Linked List*. You will implement doubly linked list and provide some operations for public access. You will implement *insert*, *delete* and *appendFirst* operations together with *getter* functions in this class.

```
template<class T>
class LinkedList {
private:
  LinkedNode<T> *head;
  LinkedNode<T> *tail;
public:
  LinkedList();

  LinkedNode<T>* getHead() const;
  LinkedNode<T>* getTail() const;
  LinkedNode<T>* getNodeAt(int idx) const;

  void insertFirst(T &data);
  void insertNode(LinkedNode<T>* node, T &data);
  void deleteNode(LinkedNode<T>* node);
};
```

Listing 2: Definition of LinkedList template class

3.2.3 Editor

This class uses *LinkedNode* and *LinkedList* template classes to implement Text Editor. Each line of this editor is stored as *Line* class and lines are connected to each other using *LinkedList* class.

This class provides an interface to modify the content of the editor. GUI part of the program, *main.cpp*, will use public functions of this class to access and alter the content of it.

In this class, there is a global cursor which stores index of the current row. This cursor shows the line at which the text should be altered. Using the cursor; you will insert/delete lines, access to them and move the cursor up and down.

Besides the cursor that stores index of the current line, there is another cursor in each line that shows the current position in the line. You can use *getCursorPosition*() function in *Line* class to get the cursor

position of that line. You will use this cursor information to insert/delete characters. Note that the cursor position in a line does not depend on the cursor position in other lines. In other words, you do not need to change the position of the cursor in lines while moving the cursor up and down. For example, let there be 2 lines in a document, first line containing 50 characters and the last line containing 25 characters. If the cursor in the first line at 30 and you move the global cursor down, the cursor in the last line should not change and stay at the position where it was.

Editor class should also keep track of inserting character, inserting line and deleting/backspacing character actions because these actions are *undoable*. History of these actions are stored in a *stack* of *Action*s and you will be pushing actions to this stack when an action is taken and popping actions from this stack to *undo* previously taken actions. For the *stack* implementation, stack template class in the standard library(std::stack) is used. *Action* class and *ActionType* enumerator can be used as you wish so long as the user actions can successfully be undone.

The Editor class also provides some *getters* to access the content of the editor. Those *getters* are used to get cursor position, line count and content of a line.

It is important to note that an editor should have at least one line at any time. There should never be an action or situation that makes the editor have less than 1 line. Therefore, in the constructor of this class, an empty line should be added.

```
class Editor {
private:
  LinkedList<Line> lines;
  std::stack<Action> history;
  int cursor;
public:
  Editor();

  void insertLine();
  void insertChar(char c);
  void del();
  void backspace();
  void undo();

  void moveCursorUp();
  void moveCursorDown();
  void moveCursorLeft();
  void moveCursorRight();

  int getCursorRow() const;
  int getCursorCol() const;
  int getLength() const;
  std::string getLine(int idx) const;
};
```

Listing 3: Definition of Editor class

3.3 GUI

GUI part contains only one file called *main.cpp*. This main file is used as the entry point of your program. It makes necessary calls from the middle layer and draws current state of the text editor to the terminal screen. Its working principle is similar to that of *vi*. It opens a new screen buffer in terminal and this new screen buffer is used to draw the state of the editor.

Important Note: This main file is given to you as a user interface to enable you test your classes easily. It only works on *∗nix* Operating Systems including *ineks*. If this main file does not work in your environment, you can write your own *main.cpp*(without GUI perhaps) and test your code with it.

Important Note 2: Ctrl+C combination is used to terminate the program.

Important Note 3: Ctrl+Z combination is used to undo the previous action.

The main program has three different modes which are listed below:

- **Default Mode:** In this mode, input is taken from the standard input and output is written to the standard output. For this mode, the call should be as follows:

```
$ make
g++ main.cpp Editor.cpp Line.cpp Action.cpp -ansi -Wall -g -O0 -pedantic -o hw1
$ ./hw1
```

- **Record Mode:** In this mode, input is taken from the standard input and the output is again written to the standard output. In addition to the default mode, your keystrokes are recorded in this one and saved to a file whose name is given as third command line argument. For this mode, the call should be as follows:

```
$ make
g++ main.cpp Editor.cpp Line.cpp Action.cpp -ansi -Wall -g -O0 -pedantic -o hw1
$ ./hw1 --record file.out
```

- **Replay Mode:** This mode is used to replay a previously created record. Previously created record file name is given as third command line argument and this file is considered as input for the program. Similar to other modes, the output is written to the standard output. For this mode, the call should be as follows:

```
$ make
g++ main.cpp Editor.cpp Line.cpp Action.cpp -ansi -Wall -g -O0 -pedantic -o hw1
$ ./hw1 --replay file.out
```

It is strongly suggested to record your inputs using Record Mode and share input and output files with your friends so that they can test your inputs with their codes.

4 Testing Your Code

In the homework files on Moodle, you will see *inputs* and *outputs* folders. There are some test cases in *inputs* folder and their expected outputs in the *outputs* folder. You can test your code with those inputs and check with the expected outputs to see if your code runs as expected or not. For this, after downloading *inputs* and *outputs* folders, you can run the following command sequence:

```
$ make
g++ main.cpp Editor.cpp Line.cpp Action.cpp -ansi -Wall -g -O0 -pedantic -o hw1
$ ./hw1 < inputs/mix.inp > mix.myout
$ diff mix.myout outputs/mix.out
```

You can also submit your source files to *Moodle* and test your work with these inputs.

5 Submission

- Submission will be done via COW.
- Create and submit *hw1.tar.gz* that contains *Action.cpp*, *Action.h*, *Editor.cpp*, *Editor.h*, *Line.cpp*, *Line.h*, *LinkedList.h* and *LinkedNode.h* files without any folder.
- Since the evaluation will be black-box, it is your responsibility to submit a package providing that the following command sequence runs successfully(textitmain.cpp will be provided by me):

```
$ tar -xzf hw1.tar.gz
$ make
g++ main.cpp Editor.cpp Line.cpp Action.cpp -ansi -Wall -g -O0 -pedantic -o hw1
$ ./hw1
```

6 Specifications

- You are not allowed to change/delete any of the class/function definition or create a new one.
- Inputs will only be composed of ASCII characters.
- Inputs will be different from the ones provided on *Moodle*. Therefore, the grade you get while testing your code may not be your actual grade that you will get during evaluation.
- Rows and columns are indexed under 0-based numbering.

7 Regulations

1. **Programming Language:** C++
2. **Libraries:** You are not allowed to use any libraries/includes except for the provided ones.
3. **Late Submission:** Please refer to *syllabus* of the course for the details.
4. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations.
5. **Newsgroup:** You must follow the newsgroup of the course for discussions and possible updates on a daily basis. For impersonal problems about the homework, sending e-mail to TAs **should not** be the first choice. Instead, you should ask your questions publicly via Newsgroup to make others know the issue and contribute.