1   Objectives

Having had first hand experience with low-level dynamic memory management, you are now ready to explore and utilize high-level turnkey alternative from the C++ standard template library (STL), which provide not only efficient memory management but also many other useful and efficient services.

To that end, this assignment will give you practice with using the STL sequential container classes such as list, vector, and string, and iterators.

2   Line Editors

In the early 1970s, before the advent of video displays and screen editors, electric typewriters were used as display devices providing a continuous printed output of a user's computer session. Multiple users were supported on the same computer, each at their own terminal. Both computers and printing terminals were very slow compared to today's standards. Meanwhile, computer programmers spent most of their work hours writing programs using **line editors**.

Typically, a line editor would prompt you for a command and then you would type in a command telling it which line you wanted displayed or edited. If you wanted to insert a line, then you would tell it that you wanted to insert a line at a particular line address and then enter that line. If you wanted to delete a line, you would have to specify the address of that line. You would repeatedly issue editing commands and then wait until the computer responded. To get a visual view of the program you were editing, you would issue a printing command and then wait until the computer responded. The wait times would add up considerably. During peak hours, programmers' editing commands could bring their editing sessions to a halt. Meanwhile, one popular line editor of the time was **ed** under Unix; it worked in silent mode, demanding minimal input, generating minimal output, and offering an extensive set of commands with forgiving syntax.

Today, line text editors are virtually useless, without practical applications. Nonetheless, the process of actually implementing a line text editor does provide not only an instructive programming experience but also plenty of opportunity to practice using the STL sequential container classes and iterators.

3   Your Task

Design and implement a line text editor, named **LineEd**[1] , without using the **new** and **delete** operators; specific requirements will be given later. We first specify the expected functionality of **LineEd**, using present tense.

---

[1]Acronym for line-oriented text editor. Note that, although **LineEd**'s command set and syntax might look a little like the commands of the **edlin** editor on DOS, or the commands of the mighty **ed** editor on Unix, **LineEd** is just a toy line editor with very limited command set and functionality

4   LineEd

LineEd is a line-oriented text editor that allows users to open, edit, and save new or existing text files.

Internally, <mark>**LineEd** always operates on a **buffer**, a place in memory where it stores a copy of the file it is editing</mark>. In addition to a **buffer**, <mark>**LineEd** uses a **clipboard**, a place in memory where cut and copied text are temporarily stored.</mark>

4.1   Staring LineEd

To start **LineEd** on a text file named **a.txt** , you type the following command in a Linux/-Mac/Windows specific shell and then presses the *return key*, which is denoted by this symbol ⏎ .

```
LineEd a.txt  ⏎
```

If, for example, the text file **a.txt** exists and has three lines, then **LineEd** reads the file contents into its **buffer**, line by line, and responds as follows:

```
"a.txt" 3 lines
Entering command mode.
?
```

Notice that **LineEd** prompts with the '**?**' symbol to indicate that it is operating in *command mode*.

When started on a nonexistent file, say, **b.txt**, **LineEd** creates an *empty* **buffer** and responds as follows:

```
"b.txt" [New File]
Entering command mode.
?
```

However, **LineEd** does not create the file **b.txt** unless and until a **w** command is entered.

Finally, when started without a filename, **LineEd** creates an *empty* **buffer** and responds as follows:

```
"?" [New File]
Entering command mode.
?
```

4.2   LineEd's Operating Modes

LineEd has two distinct operating modes.

Command mode: **LineEd** displays a '**?**' prompt to indicate it is operating in command mode. Once the return key is pressed in command mode, **LineEd** interprets the input characters as a command line.

Input mode: The **a** (append) and **i** (insert) commands put **LineEd** in input mode. **LineEd** interprets every input character as text, displaying no prompts and recognizing no commands in this mode. You can now input as many lines of text as you wish into the **buffer**, pressing the return key at the end of each line.

To put **LineEd** back in command mode, you type a period (.) on an otherwise blank line. This line is not considered part of the input text.

4.3   Command Line Syntax

LineEd command lines have a simple syntax structure:

[command symbol **] [** line address 1 **] [, [** line address 2 **]]**

The square brackets **[ ]** indicate optional parts of a command; they are used for notational purposes only and do not need to be included when entering a command. There may be any number of whitespace characters appearing before and after the optional parts of a command.

Whether or not a command requires a line range, **LineEd** allows every command to be followed by a line range. Otherwise, too many errors might ensue, resulting in an unpleasant editing session. Allowing a line range after a command, which itself may or may not be present, **LineEd** can operate silently behind the scenes, using default values for missing line addresses and command, consuming minimal input, producing minimal output, and complaining only when necessary.

4.4   Command Line Addresses

The two line addresses following a command character specify a *line range* to which the command is applied. A *line address* is either a line number, a dot character (**.**), or a dollar sign character ($), as indicated in Table 1 below:

Table 1		
Line address	Property	constraints
$	The number of the last line in the **buffer**	$ = buffer size
.	The number of the current line in the **buffer**	$1 \leq \textbf{.} \leq \$$
a line number	An integer n addressing the nth line of the **buffer**	$1 \leq n \leq \$$

4.5   LineEd Commands

Command symbols are single characters, appearing, when present, before the optional line addresses. Table 2 lists the exact syntax for each command, where the symbols x and y specify a line range, with x denoting the first line address and y the second line address, and $x \leq y$.

Table 2. **LineEd** Commands

Command	Description of **LineEd**'s Actions
a ⏎	Ignores line range, if any. Enables the LineEd input mode. Appends text into the **buffer** after the last line. The current line is set to the last line entered.
i x ⏎	Ignores line address 2, if any. Enables the LineEd input mode. Inserts text into the **buffer** at line x. The current address is set to the last line entered.
v x ⏎	Ignores line address 2, if any. Pastes text from the clipboard into the **buffer** at line x. The current address is set to the last line pasted.
d x, y ⏎	Deletes the line range x through y from the **buffer**. If there is a line after the deleted line range, then the current address is set to that line; otherwise, if there is a line before the deleted line range, then the current address is set to that line; otherwise, the **buffer** must be empty and the current line undefined.
x x, y ⏎	Cuts the line range x through y from the **buffer** into the clipboard. If there is a line after the cut line range, then the current address is set to that line; otherwise, if there is a line before the cut line range, then the current address is set to that line; otherwise, the **buffer** must be empty and the current line undefined.
j x, y ⏎	Joins the text from the specified line range together on one line at address x, concatenating the characters in the lines. Line x becomes the current line.
p x, y ⏎	Prints the line range x through y without affecting the current line address.
c x, y ⏎	Prompts for and reads the text to be changed, and then prompts for and reads the replacement text. Searches each line in the line range for an occurrence of the specified string and changes all matched strings to the replacement text. It sets the current line to the last line changed.
u ⏎	Ignores line range, if any. Moves the current line up by one line provided that there is a line above the current line; otherwise, prints the message **BOF reached** and sets the current line to the first line in the **buffer**. It prints the current line.
n ⏎	Ignores line range, if any. Moves the current line down by one line provided that there is a line after the current line; otherwise, prints the message **EOF reached** and sets the current line to the last line in the **buffer**. It prints the current line.
g x ⏎	Ignores line address 2, if any. Goes to the specified line x, meaning that it sets the current line to x and prints it.
w ⏎	Ignores line range, if any. If there is a file associated with the **buffer**, it prompts the user asking whether it is OK to replace that file with the **buffer** contents. If the answer is negative, or there is no file associated with the **buffer**, it prompts the user for the name of a file to save the **buffer** to.
q ⏎	Ignores line range, if any. Quits **LineEd**. Before quitting, however, it gives the user a last chance to save the **buffer**. If the user takes the chance, it simulates the **w** command, and then quits; otherwise, it quits, discarding **buffer** contents.

4

4.6   Command Line Rules

1. The command symbol in a command line is not case-sensitive, making no difference if it entered as uppercase or lowercase.

2. If the command symbol is omitted, then **LineED** defaults to the 'P' (print) command.

3. If the **buffer** is empty, then the command symbol must be one of the **I** (insert), **A** (append), or **Q** (quit) commands, or possibly the **V** (paste) command if **clipboard** is non-empty.

4. If the first line address option is omitted, then **LineED** defaults to the current line.

5. If the second line address option is omitted, then **LineED** defaults to the first line address.

6. If a line address is negative, then **LineED** defaults to the first line in the **buffer**.

7. If a line address exceeds the **buffer** size, then **LineED** defaults to the last line in the **buffer**.

8. Regardless of how a line range is determined, the first line address cannot exceed the second line address; otherwise, **LineEd** will swap the two line addresses to ensure that the first line address is always less than or equal to the second line address.

The rules for command lines are relaxed and forgiving, and are in place to provide a user friendly interface and to minimize challenging the user with annoying syntax errors. Any command line is allowed to end with a pair of optional line addresses, regardless of whether or not they are required.

Note that the command line rules above are based on the values of the three entities expected in a command line, namely, the command symbol, first and second line addresses, and are independent of the syntax governing the entities and of the algorithm used to parse a command line into those entities.

Table 3 below shows how a command line is interpreted; the symbol **z** represents any command from Table 2; the symbols x and y each represent either a line number, a dot character (**.**), or a dollar sign character ($) as shown in Table 1.

Table 3. Command Line Interpretation

Command Line Entered	Command Line Interpreted
z	**z.**,**.**
zx	**z**x,x
z,y	**z.**,y
zx,	**z**x,x
zx,y	**z**x,y
	p.,**.**
x	**p**x,x
,y	**p.**,y
x,	**p**x,x
x,y	**p**x,y
,	**p.**,**.**

5 Sample Editing Session

The simplest way to start **LineEd** is to run it at the command line without supplying a file name:

```
11  $   ./LineEd ⏎
12  "?" [New File]
13  Entering command mode.
14  ? p
15  File empty. Must use one of I,A,Q commands.
16  ? m
17  bad command: m
18  ?   .
19  File empty. Must use one of I,A,Q commands.
20  ? $
21  File empty. Must use one of I,A,Q commands.
22  ? i                        enter input mode and type some lines
23
24  Entering input mode.
25  line 1
26  line 2
27   .                      end of input, back to command mode
28  Entering command mode.
29  ? -10,10               silly line addresses, same as p1,$ currently
30  1: line 1
31  2> line 2
32  ? g1                   go to line 1, print it, and make it the current line
33  1> line 1
34  ? 1,$                  Print all lines
35  1> line 1
36  2: line 2
37  ? a                    append lines to the end of the buffer
38
39  Entering input mode.
40  line 3
41   .                     end of input, back to command mode
42  Entering command mode.
43  ? 1,5                  Print all lines
44  1: line 1
45  2: line 2
46  3> line 3
47  ? q                         quit
48  There are unsaved changes.
49  Do you wish to save the changes (y or n)? y
50  Enter the name of a file to write to: abc.txt
51  3 lines written to file "abc.txt"
52  bye
```

For your convenience, command lines are shown in red and my comments, which are not part of the command, are shown in brick red.

Any references to a line address refers to a line of the text being edited, not to the numbers listed outside the box in brick red color. On line 32, for example, the number 1 in command g1 refers to line number 1 in the buffer.

To indicate the current line, **LineEd** uses the symbol '>' after the line number instead of a :, as shown on lines 35 and 46.

Now, let's reopen **abc.txt** in our line editor:

```
LineEd abc.txt ⏎
```

Our driver program looks like this:

```cpp
#include <iostream>
#include <string>
#include<cstdlib> // for EXIT_FAILURE

using  std::string;
using  std::cout;
using std::cerr;
using std::endl;
#include "editor.h"

// function prototypes
void testLineEd(const string & filename);

int main(int argc, char * argv[])
{
    if (argc > 2)  // too many arguments
    {
        cerr << "Usage 1: " << argv[0] << "\n";
        cerr << "Usage 2: " << argv[0] << " filename\n";
        exit(EXIT_FAILURE);
    }

    string filename{}; // empty file name, in case argc==1
    if (argc == 2) // only one argument, a file name expected
    {
        filename = argv[1];
    };

    // Normally, we'd run the editor by uncommenting the following two lines
    //LineEd ed(filename); // create a LineED object
    //ed.run();               // run the line editor

    // test our line editor
    testLineEd(filename); // normally we'd comment out this line

    return 0;
}
```

As indicated in the comments, we would normally run our editor as shown in lines 30 and 31. At line 30, we would create an **LineEd** object passing it an input file name at construction. We would then have **LineEd** run at line 31 continuously until the user chooses to stop.

To test our line editor, and to facilitate grading it, however, we are going create a simple test script wrapped in a call of the **testLineEd(filename)** function on line 34:

abc.txt

```
line 1
line 2
line 3
```

Source code

```cpp
38  void testLineEd(const string & filename)
39  {
40  LineEd ed(filename);//create a LineED object
41  ed.runCommand("z1,$");   // bad command
42  ed.runCommand("1,$p");   // bad command
43  ed.runCommand("1$");     // bad command
44  ed.runCommand(",.p");    // bad command
```

output

```
1  "abc.txt" 3 lines
2  Entering command mode.
3  bad command: z
4  Bad address 2: $p
5  Bad address 1: 1$
6  Bad address 2: .p
```

```cpp
45
46  ed.runCommand("1,$");    // print all
47
48  // next the given invalid line range
49  // $,1 is first adjusted to 1,$,
50  // then the line range 1,$ is cut out
51  // from the buffer into the clipboard,
52  // leaving the buffer empty.
53  ed.runCommand("x$, 1");
54
55  ed.runCommand("p");// not allowed on empty buffer
```

```
7   1: line 1
8   2: line 2
9   3> line 3
10  File empty. Must use one of I,A,Q,V comm
```

```cpp
56  ed.runCommand("v");// paste ino empty buffer
57  ed.runCommand("1,$");    // print all
```

```
11  1: line 1
12  2: line 2
13  3> line 3
```

```cpp
58
59  ed.runCommand("i2");// insert at line 2
60  ed.runCommand("1,$");    // print all
```

```
14
15  Entering input mode.
16      Line 2.1
17      Line 2.2
18      Line 2.3
19   .
20  Entering command mode.
21  1: line 1
22  2:     Line 2.1
23  3:     Line 2.2
24  4>     Line 2.3
25  5: line 2
26  6: line 3
```

```
61
62  ed.runCommand("x4,5");  // cut lines 4 and 5
63  ed.runCommand("v100,1"); // same as "v1"
64  ed.runCommand("1,$");    // print all
```

```
27  1:     Line 2.3
28  2> line 2
29  3: line 1
30  4:     Line 2.1
31  5:     Line 2.2
32  6: line 3
```

```
65
66  ed.runCommand("w");// save change
```

```
33  Save changes to the file: abc.txt (y or n)? y
34  6 lines written to file "abc.txt"
```

abc.txt

```
    Line 2.3
line 2
line 1
    Line 2.1
    Line 2.2
line 3
```

```
67
68  ed.runCommand("1,$");    // print all
69  ed.runCommand("c3,6");// change 2. to two.
70  ed.runCommand("1,$");    // print all
```

```
35  1:     Line 2.3
36  2: line 2
37  3: line 1
38  4:     Line 2.1
39  5:     Line 2.2
40  6> line 3
41  change? 2.
42      to? two.
43  1:     Line 2.3
44  2: line 2
45  3: line 1
46  4:     Line two.1
47  5>     Line two.2
48  6: line 3
```

```
71
72  ed.runCommand(".");  // same as p.,.
73  ed.runCommand("n");  // down one line
74  ed.runCommand("n");  // down one line
75  ed.runCommand("1,$"); // print all
76  ed.runCommand("w");  // save buffer
```

```
49  5>     Line two.2
50  6> line 3
51  EOF reached
52  1:     Line 2.3
53  2: line 2
54  3: line 1
55  4:     Line two.1
56  5:     Line two.2
57  6> line 3
58  Save changes to the file: abc.txt (y or n)?
59  6 lines written to file "abc.txt"
```

9

abc.txt

```
    Line 2.3
line 2
line 1
    Line two.1
    Line two.2
line 3
```

```
77
78   ed.runCommand("x3"); // cut line 3
79   ed.runCommand("d1"); // del line 1
80   ed.runCommand("1,$");    // print all
```

```
60   1> line 2
61   2:    Line two.1
62   3:    Line two.2
63   4: line 3
```

```
81
82   ed.runCommand("d2,3"); // del line 2-3
83   ed.runCommand("1,$");    // print all
84   ed.runCommand("v1"); // paste at 1
85   ed.runCommand("1,$");    // print all
```

```
64   1: line 2
65   2> line 3
66   1> line 1
67   2: line 2
68   3: line 3
```

```
86
87   ed.runCommand("j2,$");   // join lines
88   ed.runCommand("1,$");    // print all
```

```
69   1: line 1
70   2> line 2line 3
```

```
89
90   ed.runCommand("j1,$");// join all
91   ed.runCommand("1,$");// print all
92   ed.runCommand("w");// save buffer
```

```
71   1> line 1line 2line 3
72   Save changes to the file: abc.txt (y or n)? y
73   1 lines written to file "abc.txt"
```

abc.txt

```
line 1line 2line 3
```

```
74
75   Entering input mode.
76   Last line
77     •
78   Entering command mode.
79   1: line 1line 2line 3
80   2> Last line
81   There are unsaved changes.
82   Do you wish to save the changes (y or n)? y
83   Save changes to the file: abc.txt (y or n)? n
84   Enter the name of a file to write to: xyz.txt
85   2 lines written to file "xyz.txt"
86   bye
```

```
93
94   ed.runCommand("a"); // append
95   ed.runCommand("$,1"); // print all
96   ed.runCommand("q"); // quit
```

xyz.txt

```
line 1line 2line 3
Last line
```

11

6   The Buffer

Since the order in which text lines are inserted into text files is important, the author of **LineEd** has to choose between the following STL sequence container classes as the underlying data structure for its **buffer**: array vector, deque, forward_list, list,

C++ arrays are quickly ruled out because of their fixed size, letting alone their lack of support for insertion or deletion of elements.

Since line editing activity typically involves insertion and deletion operations anywhere in the buffer, we are left with only two options: forward_list, and list.

Since line editing frequently involves upward and downward movement of the current line , we are left with one option: list.

```
list<string> buffer;
```

Note that the types **string** and **list**<**string**> provide highly efficient turnkey alternative to your **Line** and **LineList** classes in Assignment 1, respectively.

7   The Clipboard

The only desired operations on the clipboard are reading and overwriting its entire contents. Hence, of the five sequential containers array, vector, deque, forward_list, and list, the vector is most appropriate for the desired operations.

```
vector<string> clipboard;
```

8   Programming Requirements

- Implement two classes named **LineEd** and **Command** associated as follows:

where the dotted arrow line from class **LineEd** to class **Command** indicates that a **LineEd** object does not internally store a **Command** object. Instead, **LineEd** *uses* or *depends on* **Command** as a local variable in a member function or in the parameter list of a member function.

Command	
− command_line : string	Stores supplied command line
− status : bool	Stores validity of this command
− symbol : string	Stores the command symbol
− address1: string	Stores address 1
− address2: string	Stores address 2
+ Command(command_line : const string&) :	Ctor, sets and parses command line
+ parse(command_line : const string&) : void	Resets and parses command line
+ getSymbol() : string	Returns the command symbol
+ getAddress1() : string	Returns address 1
+ getAddress2() : string	Returns address 2
+ getStatus() : bool	Returns whether command line syntax is valid
+ setStatus(status : bool) : void	Sets status of this command

Note that class Command merely parses a given command line into a command symbol, and two line addresses and never interprets these entities; in fact, apart from the syntax of a command line, it is not even aware of the meaning of the entities within the command line. Hence, the **status** member reflects the validity of the command line syntax as a whole, as opposed to whether the command symbol is valid. In short, the class leaves interpretation of the parsed entities to the clients of the class.

The attributes involved in modeling a **LineEd** object clearly include its **buffer**, **clipboard**, **current line**, associated **file name**, whether the buffer contains unsaved contents, etc. Feel free to introduce private members (data and functions) to facilitate your modeling of the line editor.

The public interface of class **LineEd** must include a public constructor taking a file name as a parameter, a public member function running a given single command line, and a public member function running a continuous editing session.

Usage of LineEd's public members

```
LineEd editor(filename);   // create a LineEd object
editor.runCommand("p1,$"); // run a single command
editor.run();              // start an editing session
```

The private interface of class **LineEd** must include several member functions, each implementing one of the commands listed in Table 2. Again, feel free to introduce your own private member functions to facilitate your implementation of **LineEd**.

- You are not allowed to use the **new** and **delete** operators in this assignment; the idea is to recognize that it is possible to write substantial C++ programs without getting involved and entangled with dynamic memory management.

- You are not allowed to use global variables and C-style raw arrays.

9 Suggestions

- Analyze the tasks at hand, using pen and paper, and ideally away from your computer! Prepare an action plan for each task.

- Avoid writing code in large chunks thinking that you can defer testing to after completions of your code.

- You might want to start working on class **Command** first because it is independent of and simpler in functionality than class **LineEd**. Test as you write code.

  You need to have an action plan on how to parse a command line. Extracting the command symbol from a command line is rather straightforward as it can appear, if present, only at the beginning of the command line. However, parsing the line range part of a command line might be a little tricky, because a line range may have missing parts.

  You might find it easier to parse a command line after trimming out all whitespace characters in it. Since a command line is only a few characters long, it can be more efficient to directly transfer all non-white space characters from the command line to another C++ string. In any event, be sure to explore the facilities in the <string> header, including its popular family of **find** member functions.

- Learn about list iterators and about iterator operations **advance**, **distance**, **begin**, **end**, **prev**, and **next** in the <iterator> header.

- Introduce functionality to your **LineEd** class one function at a time, and test as you go, one function at a time.

  To do anything useful during an editing session, you need a non-empty buffer. So, consider implementing member functions such as **insert**, **append**, and **print** before the others. For example, to append to the end of the **buffer** your code might include elements similar to those in the following incomplete code fragment.

```
string line;
while (getline(cin, line))
{
    buffer.push_back(line);
    // other housekeeping code
}
// make sure that the current line address is set to the last line appended
```

10 Deliverables

1. Header files: **Command.h** and **LineEd.h**

2. Implementation files: **Command.cpp**, **LineEd.cpp**, **driver.cpp**

3. A **README.txt** text file (as described in the course outline).

11 Evaluation Criteria

Evaluation Criteria		
Functionality	Testing correctness of execution of your program, Proper implementation of all specified requirements, Efficiency	60%
OOP style	Encapsulating only the necessary data inside your objects, Information hiding, Proper use of C++ constructs and facilities	10%
Documentation	Description of purpose of program, Javadoc comment style for all methods and fields, comments on non-trivial pieces of code in submitted programs	10%
Presentation	Format, clarity, completeness of output, user friendly interface	10%
Code readability	Meaningful identifiers, indentation, spacing, localizing variables	10%