1 Objectives

Having had first hand experience with low-level dynamic memory management, you are now ready to explore and utilize high-level turnkey alternatives 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 iterator operations from the `<Iterator>` header.

2 Assignment Background

Back in the days before video display became popular, people used computer terminals to communicate with computers. Without a visual console, they were not able to see what was going on when they were running commands or editing files. 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.

In the early 1970s, programmers used line editors to edit their programs. Typically, a line editor would give you a prompt and then you would have to tell it which line you wanted displayed; it would then display that line, and that line only. 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 feel of what your program looked like that you were editing, you would give printing commands to reprint the edited lines 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. One of the popular line editors of the time was **ed** under Unix; it worked in silent mode, consuming minimal input and generating minimal output.

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. Hence, your assignment will be to design and implement a line text editor, which we name **led**.[1]

Here is more about **led** and its functionality, using the present tense, as if it were implemented.

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

3 led

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

Internally, **led** always operates on a **buffer**, a place in memory where it stores a copy of the file it is editing. If it is given an input file name at start, **led** loads the buffer with the content of that file, line by line; otherwise, it create an empty buffer. To write out the **buffer** to the given file, the user must give the **w** (write) command; otherwise, any changes not explicitly saved are lost.

In addition to a buffer, **led** also uses a **clipboard**, a place in memory where it stores the lines cut out of the buffer through a cut (**x**) command.

3.1 Staring led

To start **led** 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 ↵.

```
led a.txt ↵
```

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

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

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

When started on a nonexistent file, say, **b.txt**, **led** creates an *empty* **buffer** and responds as follows. However, **led** does not create the file **b.txt** unless and until a **w** command is entered.

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

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

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

3.2 led's Operating Modes

led has two distinct operating modes.

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

Input mode: The **a** (append) and **i** (insert) commands put **led** in input mode. **led** 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 **led** back in command mode, you type **Ctrl Z** under Windows and then press the return key, or type **Ctrl D** under Linux. This line is not considered part of the input text.

3.3 The Current Line

Central to **led** is the concept of the *current line*, the line most recently affected by a command, other than the print command (**p**). In fact, the concept of the *current line* is so important to **led** that it gets its own symbol (**.**), the dot character, as shown in Table 1.

3.4 The last Line

led uses the symbol(**$**) to denote the address of the last line in the buffer.

3.5 Command Line Syntax

led command lines have a simple syntax structure:

```
[line address 1][,[line address 2]][command]
```

where the optional parts of the structure are shown in brackets [ ], which may be preceded and followed by blank or tab characters; the brackets are not part of the actual command lines.

Thus, command lines each have zero, one, or two line addresses followed by an optional command. All commands are single characters. The two line addresses preceding a command specify a *line range* to which a command is applied. The value of the first line address in a line range cannot exceed the value of the the second.

A *line address* is either a line number, a dot character (**.**), or a dollar sign character ($), as indicated in Table 1 below:

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 n^{th} line of the buffer	$1 \leq n \leq \$$

Table 1

Whether or not a command requires a line range, **led** allows every command to be prefixed by a line range. Otherwise, too many errors might ensue, resulting in an unpleasant editing session. Allowing a line range before a command, which itself may or may not be present, **led** 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 it must.

Table 2 below shows how the command lines entered are interpreted, where the symbol **z** represents any of the commands listed in Table 3, and the symbols x and y represent line addresses as in Table 1.

Table 2. Command Line Interpretation

Command Line Entered	Command Line Interpreted	Constraints
x**z**	x,x**z**	$1 \leq x \leq \$$
,y**z**	**.**,y**z**	$1 \leq \textbf{.} \leq y \leq \$$
x,**z**	x,**.z**	$1 \leq x \leq \textbf{.} \leq \$$
x,y**z**	x,y**z**	$1 \leq x \leq y \leq \$$
+	1,1**+**	none
−	1,1**−**	none
x	x,x**g**	$1 \leq x \leq \$$
,y	**.**,y**p**	$1 \leq \textbf{.} \leq y \leq \$$
x,	x,**.p**	$1 \leq x \leq \textbf{.} \leq \$$
x,y	x,y**p**	$1 \leq x \leq y \leq \$$
,	**.**,**.p**	$1 \leq \textbf{.} \leq \$$
*****	1, $**p**	none

3.6 led Commands

led commands are single characters that appear at the end of command lines, after the line range, if any. Table 3 below lists the exact syntax for each command. The symbols x and y specify a line range as interpreted in Table 2, with x denoting the first address and y the second.

Commands may require zero, one, or two addresses. Commands which require zero addresses ignore the presence of address(s), if any. Commands which require only one address ignore the presence of the second address, if any.

Generally, **led** sets the current line to the last line affected by a command; however, the exact effect on the current line depends on the command and is specified in Table 3 below.

Table 3. **led** Commands

Command	Description of **led**'s Actions
x **a** ↩	In input mode, appends text in the buffer after the current line. The current address is set to the last line entered.
x **i** ↩	In input mode, inserts text in the buffer before the current line. The current address is set to the last line entered.
x,y **d** ↩	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 the current address is set to the line before the deleted line range.
x,y **x** ↩	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 the current address is set to the line before the cut line range.
x,y **r** ↩	Replaces the line range x through y with input text. Equivalent to the command line x,y**d** followed by the command line x**i**
x,y **j** ↩	Joins the line range x through y together on one line at address x, such that each line in turn is appended to line x, separated by a single space. Line x becomes the current line,
x,y **p** ↩	Prints the line range x through y without affecting the current line address.
x,y **c** ↩	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.
x **−** ↩	Moves the current line up by x lines provided that there are x lines above the current line; otherwise, prints the message **top of file reached** and sets the current line to first line in the buffer. If omitted, $x = 1$.
x **+** ↩	Moves the current line down by x lines provided that there are x lines below the current line; otherwise, prints the message **end of file reached** and sets the current line to last line in the buffer. If omitted, $x = 1$.
x **g** ↩	Goes to the specified line x, meaning that it sets the current line to x and prints it. Prints the message **invalid range** if x is invalid. If omitted, $x =$ the current line address.
w ↩	Writes out entire **buffer** to its associated file. If the **buffer** is not associated with a user named file, it prompts for and reads the name of the associated file.
q ↩	Quits **led**. 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.
* ↩	same as $1,\$$ **p** ↩
↩	same as 1**+** ↩

4 Sample Editing Session

The simplest way to start **led** is just to run it with no input file at the command line:

```
led ↵
```

Output 1

```
"?" [New File]
Entering command mode.
? 1                    move to line 1
Invalid range
? .                    print the current line
empty buffer
? $                    move to the last line
empty buffer
? p                    print the current line
empty buffer
? i                    insert input text into empty buffer
this is line 1
this is line 2
this is line 3
^Z
? *                    print out the entire buffer
1: this is line 1
2: this is line 2
3> this is line 3
? q                    before quitting, give user a last chance to save the buffer
Save changes to a file (y/n)? x
bad answer: x
Enter y for yes and n for no.
Save changes to a file (y/n)? y
Enter a file name: abc.txt
3 lines written to file: "abc.txt"
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 side numbers listed outside the output box. For example, notice that **led** uses the symbol '>' to indicate that line 3, the last line entered into the buffer, is the current line.

Now, let's reopen **abc.txt**:

```
Led abc.txt ↵
```

Output 2

```
"abc.txt" 3 lines
Entering command mode.
? *                     print out the entire buffer
1: this is line 1
2: this is line 2
3> this is line 3
```

The last line entered into the buffer becomes the current line, indicated by the symbol >.

Now, let's join lines 1 and 2:

```
? 1,2j                  join lines 1 through 2
? *                     print out the entire buffer
1> this is line 1 this is line 2
2: this is line 3
```

Now, let's join all lines:

```
? 1,$j                  join lines 1 through last
? *                     print out the entire buffer
1> this is line 1 this is line 2 this is line 3
```

Let's replace line 1 with three new lines:

```
? 1r                    replace line 1 with input text
aaaa
bbbb
cccc
^Z
? *                     print out the entire buffer
1: aaaa
2: bbbb
3> cccc
```

Now, let's insert a line before line 1 and a append line after the last line:

```
? 1i                    insert input text above line 1
1111
^Z
? p                     print the current line
1> 1111
? $a                    append input text after last line
9999
^Z
? p                     print the current line
5> 9999
? *                     print out the entire buffer
1: 1111
2: aaaa
3: bbbb
4: cccc
5> 9999
```

Now, let's cut the last two lines and paste them before the first line:

```
? 4,5x                  cut lines 4 through 5 into the clipbord
? .                     print the current line
3> bbbb
? *                     print out the entire buffer
1: 1111
2: aaaa
3> bbbb
? 1v                    paste clipboard above line 1
? .                     print the current line
2> 9999
? *                     print out the entire buffer
1: cccc
2> 9999
3: 1111
4: aaaa
5: bbbb
```

We next test the +, −, and the empty commands. First, let's attempt to move the current line, which is 2, up 5 times:

```
? 5-                    move the current line upward 5 times
top of file reached
? p                     print the current line
1> cccc
```

Notice that **led** never moves the current line beyond the top or bottom lines of the buffer; that's why the current line ended up at 1. Next, let's attempt to move down 5 times from the current line 1:

```
? *                     print out the entire buffer
1> cccc
2: 9999
3: 1111
4: aaaa
5: bbbb
? 5+                    move the current line forward 5 times
end of file reached
? p                     print the current line
5> bbbb
```

Next, let's delete the entire buffer and check out some commands:

```
? 1,$d                  delete the entire buffer
? p                     print the current line
empty buffer
? .                     print the current line
empty buffer
? 1a                    append input text after line
Invalid range
? 1,1p                  print lines 1 through 1
Invalid range
? 1i                    insert input text above line 1
Invalid range
? i                     insert input text into an empty buffer
line one
line two
line three
^Z
? p                     print the current line
3> line three
? *                     print out the entire buffer
1: line one
2: line two
3> line three
```

Notice that when the buffer is empty, any line address is invalid.

Next, let's change all the letters "e" to "E", and then change "lin" to "AC" on lines 1 and 2:

```
? *                  print out the entire buffer
1: line one
2: line two
3> line three
? 1,$c               change the entire buffer
Change what? e
     To what? E
Changed 6 occurrence(s)
? *                  print out the entire buffer
1: linE onE
2: linE two
3> linE thrEE
? 1,2c               change lines 1 through 2
Change what? lin
     To what? AC
Changed 2 occurrence(s)
? p                  print the current line
3> linE thrEE
? *                  print out the entire buffer
1: ACE onE
2: ACE two
3> linE thrEE
```

Finally, let's write a driver program for our editor, making silly errors and forcing more editing fun:

```
? 1,$r                  replace the entire buffer
    if (argc != 2)  // argc should be 2 for correct execution
    {
        cout << "usage: " << argv[0] << " <filename>\n ";
        cout << "Try again later" << endl;
        return 1;              // report an error
    }
^Z
? *                     print out the entire buffer
1:      if (argc != 2)  // argc should be 2 for correct execution
2:      {
3:          cout << "usage: " << argv[0] << " <filename>\n ";
4:          cout << "Try again later" << endl;
5:          return 1;              // report an error
6>      }
? 1i                    insert input text above line 1
int main(int argc, char * argv[])
{
^Z
? *                     print out the entire buffer
1:  int main(int argc, char * argv[])
2>  {
3:      if (argc != 2)  // argc should be 2 for correct execution
4:      {
5:          cout << "usage: " << argv[0] << " <filename>\n ";
6:          cout << "Try again later" << endl;
7:          return 1;              // report an error
8:      }
? $a                    append input text after last line
    return 0;                  // success!
}
^Z
? *                     print out the entire buffer
1:  int main(int argc, char * argv[])
2:  {
3:      if (argc != 2)  // argc should be 2 for correct execution
4:      {
5:          cout << "usage: " << argv[0] << " <filename>\n ";
6:          cout << "Try again later" << endl;
7:          return 1;              // report an error
8:      }
9:      return 0;                  // success!
10> }
```

We now need to insert code to start **led** before the successful return at line 9:

```
? 9i                    insert input text above line 9
   string filename;         // an empty filename
   filename = argv[1];      // initialize filename
   Led editor(filename);    // create an object named ed of type Led
   editor.run();            // start an editing session
^Z
? *                     print out the entire buffer
1: int main(int argc, char * argv[])
2: {
3:    if (argc != 2)  // argc should be 2 for correct execution
4:    {
5:       cout << "usage: " << argv[0] << " <filename>\n ";
6:       cout << "Try again later" << endl;
7:       return 1;            // report an error
8:    }
9:    string filename;        // an empty filename
10:    filename = argv[1];      // initialize filename
11:    Led editor(filename);    // create an object named ed of type Led
12>    editor.run();            // start an editing session
13:    return 0;                // success!
14: }
```

Now, we need to correct the errors. First, we should replace the usage line 5 and then change the condition on line 3 and as follows:

```
? 5r                    replace line 5 with input text
      cout << "usage 1: " << argv[0] << "\n ";
      cout << "usage 2: " << argv[0] << " <filename>\n ";
^Z
? 3c                    change line 3
Change what? !=
    To what? >
Changed 1 occurrence(s)
? 3c                    change line 3
Change what? be 2
    To what? be 1 or 2
Changed 1 occurrence(s)
? 3,11                  print lines 3 through 11
3>    if (argc > 2)  // argc should be 1 or 2 for correct execution
4:    {
5:       cout << "usage 1: " << argv[0] << "\n ";
6:       cout << "usage 2: " << argv[0] << " <filename>\n ";
7:       cout << "Try again later" << endl;
8:       return 1;            // report an error
9:    }
10:    string filename;        // an empty filename
11:     filename = argv[1];      // initialize filename
```

Now, at line 11, the file name must be set conditionally, so we'll replace it as follows:

```
? 11r                  replace line 11 with input text
    if (argc == 2)
    {
       filename = argv[1];      // initialize filename
    }
^Z
? *                    print out the entire buffer
1: int main(int argc, char * argv[])
2: {
3:     if (argc > 2)  // argc should be 1 or 2 for correct execution
4:     {
5:         cout << "usage 1: " << argv[0] << "\n ";
6:         cout << "usage 2: " << argv[0] << " <filename>\n ";
7:         cout << "Try again later" << endl;
8:         return 1;              // report an error
9:     }
10:     string filename;          // an empty filename
11:     if (argc == 2)
12:     {
13:        filename = argv[1];      // initialize filename
14>     }
15:      Led editor(filename);    // create an object named ed of type Led
16:      editor.run();            // start an editing session
17:      return 0;                // success!
18:   }
```

Finally, we are going to add a comment on line 11, change "ed" in the comment part of line 15 to "editor", and insert the **include** lines:

```
? 11c                  change line 11
Change what? )
     To what? )            // did the user provide a 2nd argument?
Changed 1 occurrence(s)
? 15c                  change line 15
Change what? ed of
     To what? editor of
Changed 1 occurrence(s)
? 1i                   insert input text above line 1
#include<iostream>
#include<string>
#include "led.h"
using std::cout;
using std::endl;
using std::string;
^Z
```

Now, before we print the entire file, we insert a blank line before the function header:

```
? 6,7                  print lines 6 through 7
6> using std::string;
7: int main(int argc, char * argv[])
? 7i                   insert input text above line 7

^Z
? *                    print out the entire buffer
1: #include<iostream>
2: #include<string>
3: #include "led.h"
4: using std::cout;
5: using std::endl;
6: using std::string;
7>
8: int main(int argc, char * argv[])
9: {
10:    if (argc > 2)  // argc should be 1 or 2 for correct execution
11:    {
12:       cout << "usage 1: " << argv[0] << "\n ";
13:       cout << "usage 2: " << argv[0] << " <filename>\n ";
14:       cout << "Try again later" << endl;
15:       return 1;            // report an error
16:    }
17:    string filename;        // an empty filename
18:    if (argc == 2)          // did the user provide a 2nd argument?
19:    {
20:       filename = argv[1];     // initialize filename
21:    }
22:     Led editor(filename);   // create an object named editor of type Led
23:     editor.run();           // start an editing session
24:     return 0;               // success!
25:  }
? q                    before quitting, give user a last chance to save the buffer!
Save changes to abc.txt (y/n)? y
25 lines written to file: "abc.txt"
Bye
```

5 The Buffer

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

Since line editing typically involves insertion and deletion operations anywhere in a file, **led** is left with two options: `forward_list`, and `list`.

Since line editing frequently involves upward and downward movement of the current line , **led** is left with one option: `list`.

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

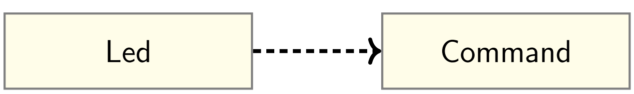
6 The Clipboard

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;
```

7 Programming Requirements

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

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

  The attributes involved in modeling a **Led** object clearly include its buffer, clipboard, current line, associated file name, whether the buffer contains unsaved contents, etc. Feel free to introduce attributes essential to your modeling of the editor.

  The public interface of class **Led** must include a public constructor that take an optional file name as a parameter and a member function that initiates an editing session, as suggested in the following code:

```
Led editor(filename);    // create a Led object
editor.run();            // start an editing session
```

The private interface of class **Led** must include several member functions, each implementing one of the commands listed in Table 3. Again, feel free to introduce any member function that can facilitate your implementation of **Led**.

- Class **Command** models a command line in terms of such attributes as the two line addresses, the command symbol, whether the command is valid, whether there is a comma in the command line, etc. Feel free to adjust the attributes to your liking. For example, you might include attributes representing default command symbol and default and ceiling values for the line addresses, initializing them at construction. The primary operation of the class should be implemented by a public member function, say **parse**, responsible for dissecting a given command line into its parts. Feel free to include other members of your choice to facilitate your work.

- 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.

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

8 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 **Led**. 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 end of the command line. However, dissecting 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 string (that's C++ string). Nonetheless, do explore the facilities in the `<string>` header, including its popular family of **find** member functions.

- Avoid getting the details of command line parsing involved in your **Led** class.
- Learn about list iterators and about iterator operations **advance**, **distance**, **begin**, **end**, **prev**, and **next** in the <iterator> header.
- Introduce functionality into your **Led** 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
```

9 Driver Program

Driver Program to test class Led

```
// Driver program to test the Led class implemented in assignment 2
#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;
#include "Led.h"
int main(int argc, char * argv[])
{   string filename;     // an empty filename
    switch (argc) {     // determine the filename
    case 1: // no file name
        break;
    case 2: filename = argv[1]; // initialize filename from argument
        break;
    default: cout << ("too many arguments - all discarded") << endl;
        break;
    }
    Led editor(filename); // create a Led named editor
    editor.run();         // run our editor
    return 0;             // report success
}
```

10 Deliverables

1. Header files: **Command.h** and **Led.h**
2. Implementation files: **Command.cpp**, **Led.cpp**, **driver.cpp**
3. A **README.txt** text file (as described in the course outline).

11 Marking scheme

60%	Program correctness: 40% `Led` 20% `Command`
15%	Program design, encapsulation, information hiding, code reuse, proper use of C++ concepts.
10%	No use of operator **new** and operator **delete**. No C-style coding and memory functions such as **malloc**, **alloc**, **realloc**, **free**, etc.
5%	Format, clarity, completeness of output
10%	Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program