Comp 5421 BB Assignment 2 June 25, 2016

Objectives

- Apply our low-level understanding of dynamic memory management to better appreciate and utilize similar but high-level turn-key software tools.
- Leverage power tools from the C++ standard template library (STL) to avoid having to reinvent the wheel every time we need a container class in a program.
- Practice programming, using the STL container classes, iterators, algorithms, and I/O processing.

Your Task

Long before video display and screen editors became popular in the 1970s, computer users used printing terminals to communicate with computers. Computer programmers spent most of their working hours with a line editor to create and manipulate their text based programs. They would typically issue editing commands to edit a line (or lines) in a file, and then they would give printing commands to reprint and check the edited line(s). That was then, and it was so slow that some programmers preferred interactive text editors that consumed minimal input and generated minimal output, saving them time and energy, giving them increased productivity, and saving them long boring wait time during I/O processing.

Today, line editors are virtually useless, without practical application. Nonetheless, they do suggest a simple and useful idea for your task in this assignment: you will implement a very simple interactive line-oriented text editor, named **led** [1], that you can use to create, edit, and save text files.

led

When supplied with a text filename , **led** initially reads the lines in the file and stores them in a temporary *somewhere* named **buffer**. Then, **led** starts the editing process on the **buffer** and not the the supplied file. To write out the **buffer** to the file you give the **w** (write) command; otherwise, any changes not explicitly saved with a **w** command are lost.

[1] Acronym for line-oriented text editor. **led**'s command set and syntax might look a little like the mighty **ed** editor for the Unix operating system, but **led** is not **ed**, not by a long shot.

For example, to run **led** on a file named **a.txt**, you type the following command and press the return key, which is denoted by ⏎:

```
led.exe a.txt ⏎
```

where the specified file **a.txt** may or may not exist. If **led** finds the file, say,

a.txt

```
this is the first line,
this is the second line, and
this is the third line.
```

it reads its contents into the **buffer**, line by line, and responds as follows:

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

led displays a ':' prompt 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 shown below. However, **led** does not create the file **b.txt** unless and until a **w** command is entered.

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

When started without a filename, or started with an empty string for the filename, **led** creates an *empty* **buffer** and responds as follows:

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

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, and then it executes that command.

Input mode: The **a** (append) and **i** (input) commands use text input mode. **led** interprets every input character as text, displaying no prompts and recognizing no commands in this mode.

You can now enter as many lines of text as you wish, pressing the return key at the end of each line. **led** gathers the input lines and places them in the **buffer**.

To end the input mode, you type the dot character '.' *alone* at the beginning of a line and press the return key, and then **led** resumes in command mode. This line is not considered part of the input text.

Here is an example of an interactive session between **led** and a user using the file **a.txt** shown on page 2. For clarity, commands and text entered by the user are shown in red, and output from **led** is shown in black. For reference, line numbers are printed in blue, and *my comments in green*.

```
1  $ led.exe a.txt                      start led on file a.txt above
2  "a.txt" 3 lines
3  Entering command mode.
4  :p print the current line, which here is the last line put in the buffer
5  this is the third line.
6  :1                          move to line 1, and make it the current line
7  this is the first line,
8  :2,3p                                         print lines 2 through 3
9  this is the second line, and
10 this is the third line.
11 :p          print the current line, which here is the last line printed
12 this is the third line.
13 :1                                                    same as line 6
14 this is the first line,
15 :2,3                                                  same as line 8
16 this is the second line, and
17 this is the third line.
18 :p                                                   same as line 11
19 this is the third line.
20 :1a                                  append input lines after line 1
```

```
21 this is a NEW second line                    enter a line of text
22 .                           Enter the dot character to exit input mode
23 :p                    print the last line that was appended to the buffer
24 this is a NEW second line
25 :1,$n                        print the entire buffer numbering each line
26 1  this is the first line,
27 2  this is a NEW second line
28 3  this is the second line, and
29 4  this is the third line.
30 :p                                                        same as line 11
31 this is the third line.
32 :2,3r                                          remove lines 2 through 3
33 this is the third line.
34 :1,$  print from line 1 through $ (last), same as the command line 1,$p
35 this is the first line,
36 this is the third line.
37 :w                          write buffer to its associated physical file
38 "a.txt" 2 lines written
39 $
```

Command Line Syntax

led command lines have a simple structure:

line address range **command**

where a *line address* is a symbol or a number that represents the number of a line in the **buffer**. A *line address range* consists of zero, one, or two line addresses, with the latter pair separated by a comma. The **command** part of a command line consists of a single character command symbol.

For example, the commands **p** on line 4, **1a** on line 20, and **2,3p** on line 8 specify zero, one, and two line addresses, respectively. Also note that the command **1** on line 6 specifies one address but no command!

As a line-editor, **led** needs to know about a range of one or more lines where it can apply any of its operations. Even if a command does not specify a line range, **led** computes the line range based on the command and context.

Whether or not a command requires a line range, **led** allows every command to have a line range before the command symbol. Otherwise, too many errors might ensue to make for a visually crowded and unpleasant editing session. By allowing a line range before a command symbol, which itself may or may not exist, **led** can better hide itself behind the scenes and complain only when it must.

A line range preceding a command symbol on a command line can consist of a maximum of two address symbols and a maximum of one separator:

Table 1: Line Range Symbols

Symbol	Description	Role
•	The current line address in the buffer. Customarily called 'dot'.	Address
$	The last line in the buffer	Address
n	A sequence of digits , $1 \leq n \leq \$$	Address
,	Separates any pair (if any) of the line address symbols above	Separator

To understand line addressing in **led** it is necessary to know that at any time there is a **current line** address maintained in **led** as the point of reference into the **buffer**. In fact, the notion of a **current line** address is so important that it gets its own symbol (•) and name (dot), as shown in Table 1.

The **current line** address is generally set to the number of the last line affected by a command; however, the exact effect on the **current line** address is presented on page 6.

For example, when the lines in a text file are read into the **buffer**, the **current line** address is set to the last line in the **buffer**, as shown in line 5 on page 3. If the **buffer** is empty, then the current line address is set 0, the address of the line before the first line; moreover, the user is forced to do one of two things: either add lines to the **buffer**, or quit **led**.

Sometimes, the **current line** address can serve as the default value for the missing address(es) in a line range of a command line. For example, the command **p** in line 4 on page 3 is equivalent to the command **3,3p**, and command **1a** on line 20 is equivalent to the command **1,1a**.

Table 2 below shows how the line range associated with an imaginary command named **z** is determined. Table 3 lists the recognized commands.

Table 2: Line Range Interpretation

Command Line Entered					Interpreted Command Line	Line Range	Constraints
,•z	•z	z	,z	•,z	•,•z	•,•	$1 \leq$ • $\leq \$$
yz					y,yz	y,y	$1 \leq y \leq \$$
,yz					•,yz	•,y	$1 \leq$ • $\leq y \leq \$$
x,z					x,•z	x,•	$1 \leq x \leq$ • $\leq \$$
x,yz					x,yz	x,y	$1 \leq x \leq y \leq \$$

Table 3: Command Line Interpretation

Command Line	Description
y **a** ↵	**led** switches to input mode, reads input lines and appends them to the buffer *after* line y. The **current line** address is set to last line entered.
y **i** ↵	**led** switches to input mode, reads input lines and inserts them to the buffer *before* line y. The **current line** address is set to last line entered. Cannot be used on empty **buffer**.
x, y **r** ↵	Removes (deletes) the line range x through y from the buffer. If there is a line after the deleted line range, then the current line address is set to this line. Otherwise the current line address is set to the line before the deleted line range.
x, y **p** ↵	Prints the addressed line range x through y. The current line address is set to the last line printed.
x, y **n** ↵	Prints the addressed line range x through y, preceding each line by its line number and a tab character. The current line address is set to the last line printed.
x, y **c** ↵	Prompts for and reads the text to be changed, and then prompts for and reads the replacement text. Searches each addressed line for an occurrence of the specified string and changes all matched strings to the replacement text.
y **u** ↵	The current line address is moved up by y lines, but never moves beyond the first line.
y **d** ↵	The current line address is moved down by y lines, but never beyond the last line.
x, y ↵	same as x, y**p**
w ↵	If the **buffer** is not associated with a user named file, it prompts for and reads a file name. Writes out entire **buffer** to its associated file.
n ↵	same as n, n**p**
. ↵	same as **.,.p**
$ ↵	same as $,$**p**
, ↵	same as 1,$**p**
= ↵	Prints the current line address.
↵	same as 1**d**

Notes:

- All letter commands are in lowercase.
- Any sequence of tab and space characters included in a command line is ignored.
- The four line address symbols in Table 1 are also listed in Table 3 as **led** commands, where none of them requires a line range.
- If a line range such as x, y is specified in a context where no line address is expected, then both x and y are ignored.

- If a line range such as x, y is specified in a context where only a single line address is expected, then x is ignored and y is used.

Programming Requirements

- **led** should use the following structure

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

  to store and edit a given text file. Note that your **LinkedList** class of assignment 1 is virtually the same as **list<string>** class!

- A class named **Command** that represents a command and in charge of parsing that command into its components. This class should provide a public member function named **parse** that takes a command line and breaks it down into its component parts: two addresses in the line range, the command symbol, a boolean flag indicating whether the command is valid, etc. Provide pertinent getters and setters as you use them. Add other members to facilitate your work.

- A class called **LineEditor** that owns the storage **buffer** and operates on it. Use pertinent attributes such as **current** for the current line, a **Command** object, etc. The only public member functions in this class are a constructor and a **run** member function, as in:

  ```
  string filename;
  // prepare filename
  LineEditor ed(filename);
  ed.run(); // runs an interactive line-editing session on filename until the user quits
  ```

  Add other members to facilitate your work.

- Add other classes to facilitate your work.

- No **new** and **delete** operators in this assignment; the idea is to learn that it is possible to write substantial C++ programs without directly dealing with dynamic memory.

- No global variables.

- As a part of this assignment, you are responsible for preparing a quick report on the similarities and differences between the standard's **list<T>**, **vector<T>** and **deque<T>** container classes.

  The member functions of these generic classes are generally categorized under the titles Iterators, Capacity, Element access, Modifiers, Operations, etc. To get credits for this part, you are required to prepare a table for each of the categories, organizing your tables as follows:

Element access

Member Function	**list<T>**	**vector<T>**	**deque<T>**
data		✓	
...	...	...	...
...	...	...	...
...	...	...	...

Modifiers

Member Function	**list<T>**	**vector<T>**	**deque<T>**
push_front	✓		✓
...	...	...	...
...	...	...	...
...	...	...	...

etc. You can use Excel spreadsheet or any other program to facilitate your work.

Suggestions

- Stay away from your PC for a few minutes, using pen and paper to analyze the problems at hand.
- Avoid writing code in large chunks thinking that you can defer testing to after completions of your code.
- Start working on the **Command** class and test as you write code. You need to have a plan of action on how to parse a command line. Extracting the command characters are rather straightforward as they can appear, if present, only after at the line range, if any. However, dissecting the line range part of a command line might be a little tricky, because a line range can have missing parts (See Table 2). For this part, you might find the popular **find** family of member functions in the **<<string>>** header helpful.
- Try introducing functionality into your **LineEditor** class one function at a time, and test as you go, one function at a time. To do anything with the **buffer**, you need to store some lines into the buffer. In fact, the only commands you can issue on an *empty* **buffer** is either add at least one line to it or quit the editor. So, consider implementing member functions such as **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;
getline(cin, line);
while (cin.good() && line != ".")
{
     buffer.push_back(line);
     getline(cin, line);
    // other housekeeping code
}
// make sure that the current line address is set to the last line appended
```

What is not shown in the code fragment above is how to keep track of the **current** line address, so that, in the case of appending to the end of the **buffer**, it always addresses the last line appended to the buffer.

An Example

To differentiate between text entered by the user and produced by **led**, the following listing uses the same color conventions as those used on page 3: user input, **led** output, and *my comments*.

```
C:\Documents\Visual Studio 2015\Projects\led\Debug>led.exe main.cpp
Unable to open file main.cpp
"main.cpp" [New File]
Entering command mode.
:p    only one of the following commands can be used on an empty buffer
error: file empty - enter 'q' to quit, 'a' to append, or 'i' to insert
:q                                                test the q command
C:\Documents\Visual Studio 2015\Projects\led\Debug>led.exe main.cpp
Unable to open file main.cpp
"main.cpp" [New File]
Entering command mode.
: i                                                insert some text
AAA
BBB
CCC
.                                       enter a dot to exit input mode
:p                 the current line should be on the last line entered
CCC
:=                                     check out the current line number
3
:1,                                    print lines 1 to the current line
AAA
BBB
CCC
:1,n                           print numbered lines 1 to the current line
1 AAA
2 BBB
3 CCC
:1i                                             insert before line 1
aaa
.                                        enter dot to exit input mode
:=                                     what's the current line number?
1
:p                                          print the current line
aaa
```

```
:1,$n                                  print the entire buffer, numbered
1 aaa
2 AAA
3 BBB
4 CCC
:3i                                          insert before line 3
bbb
.                                                exit input mode
:=                                         we should now be on line 3
3
:p                                               and should hold "bbb"
bbb
:1,$n                                           print the buffer
1 aaa
2 AAA
3 bbb
4 BBB
5 CCC
:1r                                                  remove line 1
:1,$n
1 AAA
2 bbb
3 BBB
4 CCC
:2,r                      remove lines 2 to the current line, which is 4
:1,$n                        buffer should now hold one line
1 AAA
:1,5r                              try removing an invalid line range
error:  invalid range 1 through 5
:1,1r                       empty the buffer by removing its last line
:p
error: file empty - enter 'q' to quit, 'a' to append, or 'i' to insert
:a                    let's write a test driver for our line editor.
   string filename;
   switch (argc) {
   case 1: // no file name
      break;
   case 2: // read from argument string
      filename = argv[1];
      break;
   default:
      cout << ("too many arguments - all discarded") << endl;
      filename.clear();
      break;
   }
.                    exit input mode
```

```
:1,n print lines 1 to current line, which must be the last line appended
1     string filename;
2     switch (argc) {
3     case 1: // no file name
4        break;
5     case 2: // read from argument string
6        filename = argv[1];
7        break;
8     default:
9        cout << ("too many arguments - all discarded") << endl;
10        filename.clear();
11        break;
12     }
:1i                                                  insert before line 1
int main()
{
.                                                      exit input mode
:p
{
:
=
2
:1,5n                                                   print lines 1 to 5
1  int main()
2  {
3     string filename;
4     switch (argc) {
5     case 1: // no file name
:4d                                                     move down 4 lines
:=                                                 we should now be on line 9
9
:p
      break;
:d                                                       move down 1 line
:p
   default:
:4d                                                 move down another 4 lines
:p
   }
:=                                               we should by on the last line
14
:10,n                              print numbered lines 10 to current line
10     default:
11        cout << ("too many arguments - all discarded") << endl;
12        filename.clear();
13        break;
14     }
:a                       append after the current line, that is line 14
    LineEditor ed(filename);
    ed.run();
    return 0;
}
.
:p
}
:=
18
:1p
int main()
:1c      make your program runnable from console input:  change line 1
change what? ()
    to what? (int argc, char * argv[])
:p
int main(int argc, char * argv[])
:1,5n
1  int main(int argc, char * argv[])
2  {
3     string filename;
4     switch (argc) {
5     case 1: // no file name
:6,$n
6        break;
7     case 2: // read from argument string
8        filename = argv[1];
9        break;
10     default:
11        cout << ("too many arguments - all discarded") << endl;
12        filename.clear();
13        break;
14     }
15     LineEditor ed(filename);
16     ed.run();
17     return 0;
18  }
:5,7c                    put some spaces befor any "//" in lines 5 to 7
change what? //
    to what?      //
```

```
:1,$n
1  int main(int argc, char * argv[])
2  {
3     string filename;
4     switch (argc) {
5     case 1:      // no file name
6        break;
7     case 2:      // read from argument string
8        filename = argv[1];
9        break;
10     default:
11        cout << ("too many arguments - all discarded") << endl;
12        filename.clear();
13        break;
14     }
15     LineEditor ed(filename);
16     ed.run();
17     return 0;
18  }
:16c
change what? ;
    to what? ;     // run our editor
:p
   ed.run();     // run our editor
:=
16
:100u                                            try to move up 100 lines!
BOF reached
:p                                  could not move up beyond Beginning Of File
int main(int argc, char * argv[])
:100d                                    could not move down beyond End Of File
EOF reached
:=
18:p
}
:3u
:p
   LineEditor ed(filename);
:c
change what? ;
    to what? ;     // create an editor object
:p
   LineEditor ed(filename);     // create an editor object
:5
```

```
:p
   case 1:      // no file name
:-1
error: invalid command
:2,3,4,5
error: invalid command
:hello
error: invalid command
:=
5:
:=
6
:
:p
   case 2:      // read from argument string
:=
7
:$
:=
18
:1,n
 1  int main(int argc, char * argv[])
 2  {
 3     string filename;
 4     switch (argc) {
 5     case 1:      // no file name
 6        break;
 7     case 2:      // read from argument string
 8        filename = argv[1];
 9        break;
 10     default:
 11        cout << ("too many arguments - all discarded") << endl;
 12        filename.clear();
 13        break;
 14     }
 15     LineEditor ed(filename);     // create an editor object
 16     ed.run();     // run our editor
 17     return 0;
 18  }
:q
```

```
Save changes to main.cpp (y/n)? y       must be tested
"main.cpp" 18 lines
C:\Documents\Visual Studio 2015\Projects\led\Debug> let's edit our file
again!
C:\Documents\Visual Studio 2015\Projects\led\Debug>led.exe main.cpp
"main.cpp" 18 lines
Entering command mode.
:1,$n
1  int main(int argc, char * argv[])
2  {
3     string filename;
4     switch (argc) {
5     case 1:      // no file name
6        break;
7     case 2:      // read from argument string
8        filename = argv[1];
9        break;
10     default:
11        cout << ("too many arguments - all discarded") << endl;
12        filename.clear();
13        break;
14     }
15     LineEditor ed(filename);     // create an editor object
16     ed.run();     // run our editor
17     return 0;
18  }
:q
C:\Documents\Visual Studio 2015\Projects\led\Debug>
```

Marking scheme

10%	Comparison tables as described on page 8
50%	Program correctness
20%	Proper use of pointers, dynamic memory management, and C++ concepts. No C-style memory functions such as **malloc**, **alloc**, **realloc**, **free**, etc. No C-style coding.
10%	Format, clarity, completeness of output
10%	Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program