1 Objectives

- Apply our low-level understanding of dynamic memory management to better appreciate and utilize high-level turnkey 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, C++strings, and I/O processing.

2 Your Task

Long before video display and screen editors became popular in the 1970s, computer users used printing terminals to communicate with computers. 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.

Just like today's programmers, the programmers of the 1970s spent most of their working hours with a line editor to create and manipulate their text based programs. They would typically issue an editing command and then wait until the computer responded. After editing a line (or lines) in a file, most programmers would give printing command(s) to reprint and check the edited line(s), 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. To increase productivity and decrease long wait times, they developed and used interactive line editors that consumed minimal input and generated minimal output.

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

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

3 led

led is a line-oriented text editor that allow its users to create text files and save them on their storage devices. **led** also allows its users to update existing files by deleting, changing, and inserting lines in files.

led always operates on a copy of any file it is editing, which is stored in a temporary *storage* called the **buffer**. **led** accesses the lines in its buffer by their corresponding line addresses, which are consecutive numbers, starting at line 1. When the user inserts or deletes lines of text in a file, the line addresses after the inserted or deleted lines are automatically adjusted.

To write out the **buffer** to the original file, the user gives the **w** (write) command; otherwise, any changes not explicitly saved with a **w** command are lost.

To start **led** on a text file named **a.txt** , the user types the following command in a Linux/Mac/Windows specific shell and then presses the *return key*, which is denoted by this symbol ⏎ in this assignment:

```
led a.txt ⏎
```

If, for example, the file **a.txt** exists and contains three text lines as shown here

a.txt

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

then **led** reads the file contents into its **buffer**, line by line, and responds as follows:

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

led displays a '**:**' prompt to indicate that it is now 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.
:
```

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

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

4 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** (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 a single dot character '**.**' on a line by itself and then press the return key. This line is not considered part of the input text.

5 Sample Editing Session

Here is a sample editing session using **led** on a text file named **a.txt** shown above. For clarity and reference, commands and text entered by the user are shown in red, output from **led** is shown in black, line numbers are printed in brown , and *comments in green*.

A Sample Editing Session

```
1  $ led.exe a.txt                          start led on a file named a.txt above
2  "a.txt" 3 lines     report how many lines have been copied into led's buffer
3  Entering command mode.                      start operating in 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                                  print line 1, making it the current line
7  this is the first line,
8  :2,3p                  print lines 2 through 3, making line 3 the current line
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    switch to input mode, allowing the user to insert text lines after line 1
21 this is a NEW second line                                 enter a line of text
22 .           type the dot character to exit input mode and to enter command mode
23 :p                                 print the last line inserted in 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 :
```

6 Command Line Syntax

led command lines have a simple structure:

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

where the brackets [ ] represent the optional parts of the command line.

A *command* is a single character symbol, as listed in Table 5.

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

Table 1

Line address	Meaning
.	The address of the current line in the buffer
$	The address of the last line in the buffer
a line number	An integer n such that $1 \leq n \leq$ $

In the sample editing session shown on page 4, command **p** on line 4 specifies no line addresses, command 1**a** on line 20 specifies only one line address, and command 2,3**p** on line 8 specifies two line addresses. Also note that the command line 1 on line 6 specifies one address with no command symbol present!

The two line addresses preceding a command specify a *line range*, starting at *address 1* and ending at *address 2*, to which a command is applied. **led** will use default values in place of the missing line addresses in a line range.

Whether or not a command requires a line range, **led** allows every command symbol to be prefixed by a line range. Otherwise, too many errors might ensue, resulting in an unpleasant editing session. By allowing a line range before a command symbol, which itself may or may not be present, **led** can better hide itself behind the scenes, consuming minimal input, producing minimal output, and complaining only when it must.

7 The Current Line

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

The exact effect of a command on the *current line* address is presented in Table 5.

For example, when the lines in a text file are read into the **buffer**, the *current line* address is set to the last line inserted in the **buffer**, as shown in line 5 of the sample editing session on page 4. However, if the **buffer** is or becomes empty, then the *current line* address will be undefined, forcing the user to do one of two things: either insert lines into 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 line p in line 4 on page 4 is equivalent to 3,3p, and the command line 1a on line 20 is equivalent to 1,1a. Table 2 below shows how the line range associated with such commands are calculated:

Table 2. Line range Calculation for all commands except **u** and **d**

Command Line Entered						Calculated Line Range	Constraints	**z** commands
,**z**	.**z**	**z**	,**z**	.,**z**	.,.**z**	.,.	$1 \leq . \leq \$$	**p**, **a**, **i**, **r**, **n**, **c**, **w**, **=**

Table 3 below shows how the line range associated with the commands **u** (up) and **d** (down) are calculated:

Table 3. Line range Calculation for commands **u** and **d**

Command Line Entered						Calculated Line Range	Constraints	**z** commands
,**z**	.**z**	**z**	,**z**	.,**z**	.,.**z**	1,1	$1 \leq \$$	**u**, **d**

However, when the line range in *any* command line includes at least a number or the symbol ($), the missing line address, if any, is determined as shown in Table 4 below:

Table 4. Line range Calculation for all commands missing only one line address

Command Line Entered	Calculated Line Range	Constraints
y**z**	y,y	$1 \leq y \leq \$$
,y**z**	.,y	$1 \leq . \leq y \leq \$$
x,**z**	x,.	$1 \leq x \leq . \leq \$$
x,y**z**	x,y	$1 \leq x \leq y \leq \$$

- The symbols x and y represent line numbers, or the dollar sign symbol ($).
- The symbol **z** represents any command listed in Table 5. When not specified in a command line, **z** is replaced with the default print command **p**.
- Sequences of tab and space characters in a command line are ignored.
- If a line range x, y is specified in a context where no line address is expected, then both x and y are ignored.
- If a line range x, y is specified in a context where only a single line address is expected, then x is ignored and y is used.

8 led Commands

Table 5. **led** Commands

Command Line	Function
y **a** ⏎	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** ⏎	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.
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 that line. Otherwise the current line address is set to the line before the deleted line range.
x, y **p** ⏎	Prints the line range x through y. The current line address is set to the last line printed.
x, y **n** ⏎	Prints the line range x through y, prefixing 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** ⏎	Moves the current line up by y lines, but never beyond the first line, and prints this new current line.
y **d** ⏎	Moves the current line down by y lines, but never beyond the last line, and prints this new current line.
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.
= ⏎	Prints the current line address.
⏎	same as 1**d**

9 The Buffer

Since the oder 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`.

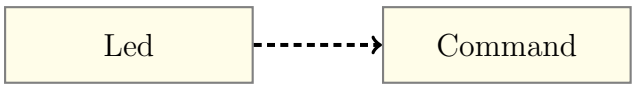
Since line editing frequently involves upward and downward movement of the current line , **led** is left with one option: `list`.

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

Note that the use of such highly optimized container classes or any other tools from the C++ STL provides a huge leverage in your programming efforts (just imagine having to reinvent `<list>` in this assignment!) All you need to do is familiarize yourself with such tools and learn how to use them.

10 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.

- Class **Command** is responsible for parsing a command line. It should include pertinent data members to represent the two line addresses, the command symbol, a boolean flag indicating whether the command is valid, a string to store relevant information (such as an error message, if any), etc. Provide pertinent getters and setters as you use them. Add other members of your choice to facilitate your work.

- Class **Led** implements **led** using a `list<string>` as its **buffer**, and includes pertinent data members and member functions. Hide all members except for the following:

  1. `Led(); //constructs an object associated with no input file`
  2. `Led(const string&); //constructs an object associated with a supplied file`
  3. `void run(); //runs an editing session until the user quits`

  Your class should include the following private member functions:

  4. A function for each of the commands listed in Table 5, taking as parameters either zero, one, or two integer line addresses, depending on the command.

5. a function `void execute(Command&)` that takes `Command` object as parameter and executes the command by delegating to appropriate command functions implemented in item 4 above.

Add other private members of your choice to facilitate your work.

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

11 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 **Command** 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 (see Tables 2-4). You might find it easier to parse a command line after trimming out all whitespace characters in it. Take advantage of the facilities in 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. Localize the use of **Command** objects in the **run()** and **execute()** member functions:

```
void Led::run()
{ string command_line;
    Command cmd;
    do
    { getline(cin, command_line); // read a command line
        cmd.parse(command_line, current_line, last_line); // parse the command line
        execute(cmd); // execute the command
    } while (cmd.symbol != 'q');
}
```

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

  To do anything during an editing session, you need to have some lines inserted in the buffer. 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
```

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

12 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 led(filename); // create an editor named led
    led.run(); // run our editor
    return 0; // done
}
```

13 Deliverables

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

14 Marking scheme

60%	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

15 A Sample Program Run

Here is another sample editing session using **led** on a new file named **driver.cpp**. For clarity and reference, commands and text entered by the user are shown in red, output from **led** is shown in black, line numbers are printed in brown , and *comments in green*.

Output

```
$ ./led driver.cpp                                              open a new file
Unable to open file driver.cpp
driver.cpp" [New File]
Entering command mode.
:i                                                          switch to insert mode
int main()

   Led led(filename);
   led.run();
   return 0 // done
}
.                                        signal end of input, switch to command mode
:p                          print the current line, which is the line inserted last
}
:1,                                              print lines 1 through current line
int main()
{
   Led led(filename);
   led.run();
   return 0 // done
}
:1,n                                  print lines 1 through current line, numbered
1  int main()
2  {
3     Led led(filename);
4     led.run();
5     return 0 // done
6  }
:u                                                                  move up 1 line
   return 0 // done                                               missing semi-colon
:c                                              change text in the current line
change what? 0
    to what? 0;
:1,$n                                    print lines 1 through last line, numbered
1  int main()
2  {
3     Led led(filename);
4     led.run();
5     return 0; // done
6  }
:1                                                                  move to line 1
int main()
:i                                                      insert before current line



#include<iostream>
#include<string?
using std::cout;
using std::endl;
using std::string;
#include "Led.h"

.                                                  end input mode, back to command mode
:d                             already on the last line.  try to move down by 1 line
EOF reached
```

```
}
:1,$                                                print the entire file contents



#include<iostream>
#include<string?
using std::cout;
using std::endl;
using std::string;
#include "Led.h"

int main()
{
   Led led(filename);
   led.run();
   return 0; // done
}
:1                                                                  move to line 1

:d                                                          move down 1 line to line 2

:p                                        print the current line, which is a blank line

:=                                                       print the current line number
2
:-----                                                  enter some invalid command line
invalid line address: ------
:1,$n                                                    print the entire file numbered
1
2
3
4  #include<iostream>
5  #include<string?
6  using std::cout;
7  using std::endl;
8  using std::string;
9  #include "Led.h"
10
11  int main()
12  {
13     Led led(filename);
14     led.run();
15     return 0; // done
16  }
:=                                                       print the current line number
16
:5c                                                    in line 5, change ?  to >
change what? ?
    to what? >
:p                                                          print the current line
#include<string>
:11c                                                                change line 11
change what? ()
    to what? (int argc, char * argv[])
```

```
:p                                                    print the current line
int main(int argc, char * argv[])
:=                                                 print the current line number
11
:1,3r                                                remove lines 1-3, inclusive
:p                                                    print the current line
#include<iostream>
:=                                                 print the current line number
1
:,$                                              print current line to last line
#include<iostream>
#include<string>
using std::cout;
using std::endl;
using std::string;
#include "Led.h"

int main(int argc, char * argv[])
{
   Led led(filename);
   led.run();
   return 0; // done
}
:=                                                 print the current line number
13
:1,n                                       print current line to last line, numbered
1  #include<iostream>
2  #include<string>
3  using std::cout;
4  using std::endl;
5  using std::string;
6  #include "Led.h"
7
8  int main(int argc, char * argv[])
9  {
10     Led led(filename);
11     led.run();
12     return 0; // done
13  }
:1i                                                      insert before line 1
// Driver program to test the
// Led class implemented in assignment 2
.
:1,5n                                          print lines 1 through 5, numbered
1  // Driver program to test the
2  // Led class implemented in assignment 2
3  #include<iostream>
4  #include<string>
5  using std::cout;
:10,$n                                          print lines 10 through last line
```

```
10  int main(int argc, char * argv[])
11  {
12     Led led(filename);
13     led.run();
14     return 0; // done
15  }
:1,n                                  print lines 1 through current line numbered
1  // Driver program to test the
2  // Led class implemented in assignment 2
3  #include<iostream>
4  #include<string>
5  using std::cout;
6  using std::endl;
7  using std::string;
8  #include "Led.h"
9
10  int main(int argc, char * argv[])
11  {
12     Led led(filename);
13     led.run();
14     return 0; // done
15  }
:11a                                                        append after line 11
   string filename; // an empty line
   // determine the filename

   switch(argc)
   {
   case 1:  // no file name
   brake; // spelling error

   case 2:  // read from argument string
   filename = argv[1]; // initialize filename
      brake;

   default:
   cout << "too many arguments - all discarded") << endl;
      brake;
   }
.
:1,$n                                                  print all lines numbered
1  // Driver program to test the
2  // Led class implemented in assignment 2
3  #include<iostream>
4  #include<string>
5  using std::cout;
6  using std::endl;
7  using std::string;
8  #include "Led.h"
9
```

```
10  int main(int argc, char * argv[])
11  {
12      string filename; // an empty line
13      // determine the filename
14
15      switch(argc)
16      {
17          case 1: // no file name
18              brake; // spelling error
19
20          case 2: // read from argument string
21              filename = argv[1]; // initialize filename
22              brake;
23
24          default:
25              cout << "t>o many arguments - all discarded") << >ndl;
26              brake;
27      }
28      Led led(filename);
29      led.run();
30      return 0; // done
31  }
:18,26c                                   in lines 18-26, change brake to break
change what? brake
    to what? break
:18,26                                                print lines 18 through 20
          break; // spelling error

       case 2: // read from argument string
          filename = argv[1]; // initialize filename
          break;

       default:
          cout << "t>o many arguments - all discarded") << >ndl;
          break;
:18c                                   remove '// spelling error' in line 18
change what? // spelling error
    to what?
:,n                                                   print current line numbered
18            break;
:.n                                                   print current line numbered
18            break;
:.,.                                                  print current line numbered
          break;
:=                                                           what line are we at?
18
:2d                                              move current line down by 2 lines
       case 2: // read from argument string
:8d                                              move current line down by 2 lines
   Led led(filename);
:$                                                             move to last line
}
```

```
:1,n                          print lines 1 through the current line, numbered
1  // Driver program to test the
2  // Led class implemented in assignment 2
3  #include<iostream>
4  #include<string>
5  using std::cout;
6  using std::endl;
7  using std::string;
8  #include "Led.h"
9
10  int main(int argc, char * argv[])
11  {
12     string filename; // an empty line
13     // determine the filename
14
15     switch(argc)
16     {
17        case 1: // no file name
18           break;
19
20        case 2: // read from argument string
21           filename = argv[1]; // initialize filename
22           break;
23
24        default:
25           cout << "t>o many arguments - all discarded") << >ndl;
26           break;
27     }
28     Led led(filename);
29     led.run();
30     return 0; // done
31  }
:w                                                    write buffer to file
"driver.cpp" 31 lines written
:q                                                        quit the editor
quitting led . . . bye.
$ ./led driver.cpp                                  edit the same file again
"driver.cpp" 31 lines
Entering command mode.
:2a                                                    append after line 2
#include<cstdlib>
.
:1,n                              print lines 1 through current line, numbered
1  // Driver program to test the
2  // Led class implemented in assignment 2
3  #include<cstdlib>
```

```
:1,$n                                                    print all lines numbered
1  // Driver program to test the
2  // Led class implemented in assignment 2
3  #include<cstdlib>
4  #include<iostream>
5  #include<string>
6  using std::cout;
7  using std::endl;
8  using std::string;
9  #include "Led.h"
10
11  int main(int argc, char * argv[])
12  {
13     string filename; // an empty line
14     // determine the filename
15
16     switch(argc)
17     {
18        case 1: // no file name
19           break;
20
21        case 2: // read from argument string
22           filename = argv[1]; // initialize filename
23           break;
24
25        default:
26           cout << ">oo many arguments - all discarded") <<>endl;
27           break;
28     }
29     Led led(filename);
30     led.run();
31     return 0; // done
32  }
:100u       try to move up the current line by 100 lines, which is beyond line 1
BOF reached
// Driver program to test the
:100d  try to move down the current line by 100 lines, which is beyond last line
EOF reached
}                                                        this is the last line
:=                                                       Which line are we at?
32
:4                                                                move to line 4
#include<iostream>
:                                        press the enter key to move down 1 line
#include<string>
:                                        press the enter key to move down 1 line
using std::cout;
:                                        press the enter key to move down 1 line
using std::endl;
:                                        press the enter key to move down 1 line
using std::string;
:                                        press the enter key to move down 1 line
#include "Led.h"
:=                                                       Which line are we at?
9
```

```
:u                                                          move up 1 line
using std::string;
:u                                                          move up 1 line
using std::endl;
:=                                                      Which line are we at?
7
:q                                                          quit the editor
Save changes to "driver.cpp" (y/n)?  yes
invalid answer: yes
enter y for yes and n for no.
Save changes to "driver.cpp" (y/n)? y
"driver.cpp" 32 lines written
quitting led . . . bye.
```