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](https://en.wikipedia.org/wiki/Computer_terminal) 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](https://en.wikipedia.org/wiki/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 le, 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 in-crease 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. Specically, you will implement a simple interactive line-oriented text editor, named led 1, that you can use to create, edit, and save text les.

1Acronym for line-oriented text editor. Although led’s command set and syntax might look a little like that of the mighty [ed](https://en.wikipedia.org/wiki/Ed_text_editor) editor for the Unix operating system, led is a toy line editor with very limited command set and functionality.

1

3 led

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

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

To write out the buer to the original le, 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 le named a.txt , the user types the following command in a Linux/Mac/Windows specic shell and then presses the return key, which is denoted by this symbol in this assignment:

led a.txt

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

a.txt

this is the this is the this is the

first line, second line, and third line.

then led reads the le contents into its buer, 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 le, say, b.txt, led creates an empty buer and responds as shown below. However, led does not create the le b.txt unless and until a w command is entered.

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

2

Finally, when started without a lename, led creates an empty buer and responds as follows:

"?" [New Entering :

File]

command mode.

4 Operating Modes

led has two distinct operating modes.

Command mode: led displays a ’:’ prompt to indicate it is operating in com-mand 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 buer, pressing the return key at the end of each line.

To put led back in command mode, you type a single dot char-acter ’.’ on a line by itself and then press the return key. This line is not considered part of the input text.

3

5 Sample Editing Session

Here is a sample editing session using led on a text le 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 :

4

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 dened in Table 1 below:

Table 1

|  |  |
| --- | --- |
| Line address | Meaning |
| . | The address of the current line in the buer |
| $ | The address of the last line in the buer |
| a line number | An integer n such that 1 n $ |

In the sample editing session shown on page 4, command p on line 4 species no line addresses, command 1a on line 20 species only one line address, and command 2,3p on line 8 species two line addresses. Also note that the command line 1 on line 6 species 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 prexed 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 aected 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 eect of a command on the current line address is presented in Table 5.

5

For example, when the lines in a text le are read into the buer, the current line address is set to the last line inserted in the buer, as shown in line 5 of the sample editing session on page 4. However, if the buer is or becomes empty, then the current line address will be undened, forcing the user to do one of two things: either insert lines into the buer, 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 . $ | 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 $ | 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 |
| yz | y,y | 1 y $ |
| ,yz | .,y | 1 . y $ |
| x,z | x,. | 1 x . $ |
| x,yz | x,y | 1 x y $ |

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 specied 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 specied in a context where no line address is expected, then both x and y are ignored.

If a line range x;y is specied in a context where only a single line address is expected, then x is ignored and y is used.

6

8 led Commands

Table 5. led Commands

|  |  |
| --- | --- |
| Command Line | Function |
| y a | Switches to input mode, reads input lines and appends them to the buer 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 buer 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 buer. 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, prexing 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 ad-dressedlineforanoccurrenceofthespeciedstringandchanges all matched strings to the replacement text. |
| y u | Moves the current line up by y lines, but never beyond the rst 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 buer to its associated le. If the buer is not associated with a user named le, it prompts for and reads the name of the associated le. |
| = | Prints the current line address. |
|  | same as 1d |

9 The Buer

Since the oder in which text lines are inserted in text les is important, led has to choose between one of the following STL sequence container classes as the underlying data structure for its buer: array, vector, deque, forward list, and list.

7

Since line editing typically involves insertion and deletion operations anywhere in a le, 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 eorts (just imagine having to reinvent [<list>](http://www.cplusplus.com/reference/list/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:

|  |  |  |
| --- | --- | --- |
| Led |  | Command |
|  |

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 sym-bol, a boolean ag 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 buer, 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.

8

5. a function void execute(Command&) that takes Command object as param-eter 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 rst 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 nd 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 nd 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’); }

9

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 buer. So, consider implementing member functions such as append and print before the others. For example, to append to the end of the buer 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](http://www.cplusplus.com/reference/list/list/) iterators and about iterator operations advance, distance, begin, end, prev, and next in the [<iterator>](http://www.cplusplus.com/reference/iterator/) header.

12 Driver Program

Driver Program to test class Led

1 // Driver program to test the Led class implemented in assignment 2 2 #include <iostream>

3 #include <string> 4 using std::cout; 5 using std::endl;

6 using std::string; 7 #include "Led.h"

8 int main(int argc, char \* argv[])

9 { string filename; // an empty filename

10 switch (argc) { // determine the filename 11 case 1: // no file name

12 break;

13 case 2: filename = argv[1]; // initialize filename from argument 14 break;

15 default: cout << ("too many arguments - all discarded") << endl; 16 break;

17 }

18 Led led(filename); // create an editor named led 19 led.run(); // run our editor

20 return 0; // done 21 }

10

15 A Sample Program Run

Here is another sample editing session using led on a new le 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.

11

Output

1 $ ./led driver.cpp

2 Unable to open file driver.cpp 3 driver.cpp" [New File]

4 Entering command mode. 5 :i

6 int main()

7

open a new file

switch to insert mode

8 Led led(filename); 9 led.run();

10 return 0 // done 11 g

12 . signal end of input, switch to command mode 13 :p print the current line, which is the line inserted last 14 }

15 :1, print lines 1 through current line 16 int main()

17 {

18 Led led(filename); 19 led.run();

20 return 0 // done 21 }

22 :1,n print lines 1 through current line, numbered 23 1 int main()

24 2 {

25 3 Led led(filename); 26 4 led.run();

27 5 return 0 // done 28 6 }

29 :u move up 1 line 30 return 0 // done missing semi-colon 31 :c change text in the current line 32 change what? 0

33 to what? 0;

34 :1,$n print lines 1 through last line, numbered 35 1 int main()

36 2 {

37 3 Led led(filename); 38 4 led.run();

39 5 return 0; // done 40 6 }

41 :1 move to line 1 42 int main()

43 :i insert before current line

44

45

46

47 #include<iostream> 48 #include<string?

49 using std::cout; 50 using std::endl;

51 using std::string; 52 #include "Led.h"

53

54 . 55 :d

56 EOF reached

end input mode, back to command mode already on the last line. try to move down by 1 line

57 }

58 :1,$

59

60

61

62 #include<iostream> 63 #include<string?

64 using std::cout; 65 using std::endl;

66 using std::string; 67 #include "Led.h"

68

69 int main() 70 {

71 Led led(filename); 72 led.run();

73 return 0; // done 74 }

75 :1

76

77 :d 78

79 :p

80

print the entire file contents

move to line 1

move down 1 line to line 2

print the current line, which is a blank line

81 := 82 2

83 :-----

84 invalid line address: ------85 :1,$n

86 1 87 2 88 3

89 4 #include<iostream> 90 5 #include<string?

91 6 using std::cout; 92 7 using std::endl;

93 8 using std::string; 94 9 #include "Led.h"

95 10

96 11 int main() 97 12 {

98 13 Led led(filename); 99 14 led.run();

100 15 return 0; // done 101 16 }

102 := 103 16 104 :5c

105 change what? ? 106 to what? > 107 :p

108 #include<string> 109 :11c

110 change what? ()

111 to what? (int argc, char \* argv[]) 13

print the current line number

enter some invalid command line

print the entire file numbered

print the current line number

in line 5, change ? to >

print the current line

change line 11

112 :p

113 int main(int argc, char \* argv[]) 114 :=

115 11

116 :1,3r 117 :p

118 #include<iostream> 119 :=

120 1 121 :,$

122 #include<iostream> 123 #include<string> 124 using std::cout; 125 using std::endl; 126 using std::string; 127 #include "Led.h"

128

print the current line

print the current line number

remove lines 1-3, inclusive print the current line

print the current line number

print current line to last line

129 int main(int argc, char \* argv[]) 130 {

131 Led led(filename); 132 led.run();

133 return 0; // done 134 }

135 := print the current line number 136 13

137 :1,n print current line to last line, numbered 138 1 #include<iostream>

139 2 #include<string> 140 3 using std::cout; 141 4 using std::endl; 142 5 using std::string; 143 6 #include "Led.h" 144 7

145 8 int main(int argc, char \* argv[]) 146 9 {

147 10 Led led(filename); 148 11 led.run();

149 12 return 0; // done 150 13 }

151 :1i insert before line 1 152 // Driver program to test the

153 // Led class implemented in assignment 2 154 .

155 :1,5n print lines 1 through 5, numbered 156 1 // Driver program to test the

157 2 // Led class implemented in assignment 2 158 3 #include<iostream>

159 4 #include<string> 160 5 using std::cout;

161 :10,$n print lines 10 through last line

14

162 10 int main(int argc, char \* argv[]) 163 11 {

164 12 Led led(filename); 165 13 led.run();

166 14 return 0; // done 167 15 }

168 :1,n print lines 1 through current line numbered 169 1 // Driver program to test the

170 2 // Led class implemented in assignment 2 171 3 #include<iostream>

172 4 #include<string> 173 5 using std::cout; 174 6 using std::endl; 175 7 using std::string; 176 8 #include "Led.h" 177 9

178 10 int main(int argc, char \* argv[]) 179 11 {

180 12 Led led(filename); 181 13 led.run();

182 14 return 0; // done 183 15 }

184 :11a append after line 11 185 string filename; // an empty line

186 // determine the filename

187

188 switch(argc) 189 f

190 case 1: // no file name 191 brake; // spelling error

192

193 case 2: // read from argument string

194 filename = argv[1]; // initialize filename 195 brake;

196

197 default:

198 cout << "too many arguments - all discarded") << endl; 199 brake;

200 g 201 .

202 :1,$n print all lines numbered 203 1 // Driver program to test the

204 2 // Led class implemented in assignment 2 205 3 #include<iostream>

206 4 #include<string> 207 5 using std::cout; 208 6 using std::endl; 209 7 using std::string; 210 8 #include "Led.h" 211 9

15

212 10 int main(int argc, char \* argv[]) 213 11 {

214 12 string filename; // an empty line 215 13 // determine the filename

216 14

217 15 switch(argc) 218 16 {

219 17 case 1: // no file name

220 18 brake; // spelling error 221 19

222 20 case 2: // read from argument string

223 21 filename = argv[1]; // initialize filename 224 22 brake;

225 23

226 24 default:

227 25 cout << "t>o many arguments - all discarded") << >ndl; 228 26 brake;

229 27 }

230 28 Led led(filename); 231 29 led.run();

232 30 return 0; // done 233 31 }

234 :18,26c in lines 18-26, change brake to break 235 change what? brake

236 to what? break

237 :18,26 print lines 18 through 20 238 break; // spelling error

239

240 case 2: // read from argument string

241 filename = argv[1]; // initialize filename 242 break;

243

244 default:

245 cout << "t>o many arguments - all discarded") << >ndl; 246 break;

247 :18c remove ’// spelling error’ in line 18 248 change what? // spelling error

249 to what?

250 :,n print current line numbered 251 18 break;

252 :.n print current line numbered 253 18 break;

254 :.,. print current line numbered 255 break;

256 := what line are we at? 257 18

258 :2d move current line down by 2 lines 259 case 2: // read from argument string

260 :8d move current line down by 2 lines 261 Led led(filename);

262 :$ move to last line 263 }

16

264 :1,n print lines 1 through the current line, numbered 265 1 // Driver program to test the

266 2 // Led class implemented in assignment 2 267 3 #include<iostream>

268 4 #include<string> 269 5 using std::cout; 270 6 using std::endl; 271 7 using std::string; 272 8 #include "Led.h" 273 9

274 10 int main(int argc, char \* argv[]) 275 11 {

276 12 string filename; // an empty line 277 13 // determine the filename

278 14

279 15 switch(argc) 280 16 {

281 17 case 1: // no file name 282 18 break;

283 19

284 20 case 2: // read from argument string

285 21 filename = argv[1]; // initialize filename 286 22 break;

287 23

288 24 default:

289 25 cout << "t>o many arguments - all discarded") << >ndl; 290 26 break;

291 27 }

292 28 Led led(filename); 293 29 led.run();

294 30 return 0; // done 295 31 }

296 :w write buffer to file 297 "driver.cpp" 31 lines written

298 :q quit the editor 299 quitting led . . . bye.

300 $ ./led driver.cpp edit the same file again 301 "driver.cpp" 31 lines

302 Entering command mode.

303 :2a append after line 2 304 #include<cstdlib>

305 .

306 :1,n print lines 1 through current line, numbered 307 1 // Driver program to test the

308 2 // Led class implemented in assignment 2 309 3 #include<cstdlib>

17

310 :1,$n print all lines numbered 311 1 // Driver program to test the

312 2 // Led class implemented in assignment 2 313 3 #include<cstdlib>

314 4 #include<iostream> 315 5 #include<string> 316 6 using std::cout; 317 7 using std::endl; 318 8 using std::string; 319 9 #include "Led.h" 320 10

321 11 int main(int argc, char \* argv[]) 322 12 {

323 13 string filename; // an empty line 324 14 // determine the filename

325 15

326 16 switch(argc) 327 17 {

328 18 case 1: // no file name 329 19 break;

330 20

331 21 case 2: // read from argument string

332 22 filename = argv[1]; // initialize filename 333 23 break;

334 24

335 25 default:

336 26 cout << ">oo many arguments - all discarded") <<>endl; 337 27 break;

338 28 }

339 29 Led led(filename); 340 30 led.run();

341 31 return 0; // done 342 32 }

343 :100u try to move up the current line by 100 lines, which is beyond line 1 344 BOF reached

345 // Driver program to test the

346 :100d try to move down the current line by 100 lines, which is beyond last line 347 EOF reached

348 } this is the last line 349 := Which line are we at? 350 32

351 :4 move to line 4 352 #include<iostream>

353 : press the enter key to move down 1 line 354 #include<string>

355 : press the enter key to move down 1 line 356 using std::cout;

357 : press the enter key to move down 1 line 358 using std::endl;

359 : press the enter key to move down 1 line 360 using std::string;

361 : press the enter key to move down 1 line 362 #include "Led.h"

363 := Which line are we at? 364 9

18

365 :u

366 using std::string; 367 :u

368 using std::endl; 369 :=

370 7 371 :q

372 Save changes to "driver.cpp" (y/n)? yes 373 invalid answer: yes

374 enter y for yes and n for no.

375 Save changes to "driver.cpp" (y/n)? y 376 "driver.cpp" 32 lines written

377 quitting led . . . bye.

move up 1 line

move up 1 line

Which line are we at?

quit the editor

19