ECS 36C: Programming Assignment #3

Instructor: Aaron Kaloti

Winter Quarter 2022

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: To the end of the subsection "Miscellaneous Remarks", I added information about what you can assume regarding use of whitespace in the commands.
- v.3:
  - Autograder details.
  - Added the compilation command (in the autograder section).
  - You may assume that the undoing of an insertion will never cause the cursor to be out of bounds. (That is, the autograder won't cause something like this.)
- v.4: Deadline pushed back to the night of Friday, 02/18.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Friday, 02/18. Gradescope will say 12:30 AM on Saturday, 02/19, due to the "grace period" (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

3 Purpose of This Assignment

- To show you why the ECS 36C course description says, "Extensive programming."
- To give you experience with a project that takes *at least* 350-450 lines of code and that really tests your problem-solving skills and ability to handle a lot of details/specifications (like any real-world project would have).
- To give you a practical example of a reason to use a stack.

4 Reference Environment

The autograder, which is talked about at the end of this document, will compile and run your code in a Linux environment. That means that you should make sure your code compiles and behaves properly in a sufficiently similar environment. **The CSIF is one such environment.** Each student can remotely access the CSIF, and now that learning is back in person, the CSIF computers can be physically accessed in the Kemper basement. I talk more about the CSIF in the syllabus.

Do not assume that because your code compiles on an *insufficiently* similar environment (e.g. directly on your Mac laptop), it will compile on the Linux environment used by the CSIF.

You should avoid causes of undefined behavior in your code, such as uninitialized variables. If you have things like this in your code, then your code could end up generating the wrong answer when autograded, even if it generated the right answer when you tested it in a sufficiently similar environment.

5 Ayayron™: An Editor That No One in Their Right Mind Would Ever Use

Filename: `editor.cpp`

In this assignment, you will implement a pretty terrible text editor in a C++ file called `editor.cpp`. This is the only file that you will submit.

Although the parts below come in a certain order, you don't have to do them in the exact order shown. Needless to say, some parts have to be done (or at the very least, *should* be done) before others; for example, it wouldn't make much sense to implement the window scrolling before you implement the reading of the input file's contents. I would encourage you to read this entire document before starting, because it is possible that some of the later parts could affect how you do the earlier parts. That is, you do not want to run into a situation in which, in a later part, you have to rewrite code you wrote in an earlier part.

5.1 Miscellaneous Remarks

Don't forget that your output must match mine *exactly*. The only exception is trailing whitespace, which the autograder removes.

As with the cursor list assignment, it will still be possible to get credit even if you do not finish the entire assignment. Moreover, not all parts depend on the parts before it. What this means is that, for example, if you cannot get downward scrolling to completely work, you could still get some of the test cases for undo/redo correct. The order of the parts is not the only order in which you can do them.

You will be penalized if your program has any memory leaks or fails to close any file. However, these should be easy things to avoid if you use the typical C++ features/libraries (as opposed to the C ones) that follow RAII principles.

You do not need to worry about the user entering invalid inputs. For instance, later on, you will support the ability for the user to enter 'r' in order to trigger a redo operation. You do not need to worry about the user/autograder trying to trick your code with inputs like "ribbon" or "r o c k". The autograder will not assess your program's response to invalid inputs. However, in my own implementation, I made sure my program didn't die/crash in the event of an invalid input, because I sometimes accidentally entered an invalid input while trying to test my code.

For any input that involves an initial character or word (e.g. `s 3`, `w 4`, `save output.txt`), you may assume that there will always be one space between the initial character/word and whatever comes after. For instance, you will never see the input `w     4`. This does not apply to spaces in what is inserted during an insertion operation. For instance, `i A     B` is valid, even though there are five spaces between the 'A' and the 'B'.

5.2 Part #1 - Reading the Input File's Contents

Your editor will be passed the name of the file to read as a command-line argument.

If your program is given too few or too many command-line arguments, then it should print out an appropriate message and return the appropriate exit code, as shown below. *Everything your program prints – including error messages like the below – should be printed to standard output, NOT standard error. You should not print to standard error at any point in this assignment.*

```
$ ./editor
Too few command-line arguments.
$ echo $?
1
$ ./editor a b
Too many command-line arguments.
$ echo $?
1
```

Your editor should read in the contents of the file. You will need to decide how best to store these contents, and that decision could be influenced by later parts. Since a file's contents are better read line-by-line rather than word-by-word, you may find `std::getline()` more useful than `>>` for reading from the file. Be careful about using *both* `std::getline()` and `>>` on the same stream, whether a file stream or the standard input stream (`std::cin`). As an example of what can go wrong if you use both of them: if you reach the end of a line with `>>` and then use `std::getline()` once, the latter will read an empty string, not the next line.

If any of the below occur, the editor should print an appropriate message and terminate, returning the appropriate exit code.

- The file cannot be opened. (Do not worry about the specific reason, e.g. the file does not exist, insufficient permissions, etc.)
- The file has more than 30 lines.
- At least one line in the file is more than 20 characters long. (The implicit newline character at the end of each line in the text file does not count.)

Below are examples showing how your program should react in the above situations. Note that you do not need to store the input files in a folder called `example_input_files`; I did that as part of my own file organization.

```
$ ./editor nonexistent_file
Failed to open file: nonexistent_file
$ echo $?
2
$ wc -l example_input_files/too_many_lines.txt
31 example_input_files/too_many_lines.txt
$ ./editor example_input_files/too_many_lines.txt
File example_input_files/too_many_lines.txt has too many lines.
$ echo $?
2
$ cat example_input_files/wide_line.txt
ab
cd
123456789012345678901
ef
gh
$ ./editor example_input_files/wide_line.txt
File example_input_files/wide_line.txt has at least one too long line.
$ echo $?
2
```

The input file will never be empty.

5.3 Part #2 - The Main Loop of Your Program

The main loop of your program should do the following:

- Display the current buffer contents[1].
- Prompt the user for a command.
- React to the command.

[1] I don't want to say "current file contents" because, if a modification/insertion is made and a save operation is not done, then the output file's contents will be out-of-sync with what the editor shows.

Whenever the edited file's contents are displayed, it will be within a window that is 20 characters wide and 10 characters tall. Through this window, we see only a part of the current buffer contents at any given time. Initially, the window should start at line #1. In a later part, you will allow the window to be scrolled up and down.

The only command that your program should support at this point is quitting by entering 'q'. (You can see this being entered after "Enter command: " is printed in the below.) This should result in the printing of a message (see below) and the termination of the program.

Below are two examples of how your program should behave.

```
$ wc -l example_input_files/input1.txt
27 example_input_files/input1.txt
$ ./editor example_input_files/input1.txt
     *
     12345678901234567890
*  1|abcdef
   2|ghi
   3|jklm
   4|jkj
   5|jklasj
   6|jklajs;lkfj
   7|qiowuioj
   8|lkjw;lkj
   9|qwklejklj;c
  10|jkaljsoiu
     12345678901234567890
Enter command: q

Goodbye!
$ wc -l example_input_files/small.txt
5 example_input_files/small.txt
$ ./editor example_input_files/small.txt
     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: q

Goodbye!
$ echo $?
0
```

Just so it's clear, there are two whitespaces between the asterisk and the number 1, in the case of the asterisk on the left column. If the asterisk were on line number 10, then there would be one whitespace between the asterisk and the number 10. In the second example, the vertical bar is not printed after the line number for line #6 onwards because `small.txt` does not have more than five lines.

Line numbers, as well as rows containing `12345678901234567890`, are printed above and below the window's contents, for convenience. (It is sometimes useful to know which column you are on in an editor, even if that editor is terrible.) You can also see two asterisks in the above output as well. Those asterisks mark the current location of the cursor. In later parts, you will allow the cursor to be moved and allow the user to make insertions starting at the current location of the editor.

5.4 Part #3 - Saving the File

If the user enters "save", you may assume it will be followed by a filename, and your program should save the current buffer contents to that file. You may assume that there will never be issues (e.g. permission issues) with opening/creating the file to write to.

Below is an example of how your program should behave.

```
$ ./editor example_input_files/small.txt
     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: save aaron_is_cool.txt

     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: q

Goodbye!
$ cat aaron_is_cool.txt
Line 1
Line 2
Line 3
Line 4
Line 5
```

5.5 Part #4 - Moving the Cursor Left/Right

If the user types ‘a’ or ‘d’, then – assuming it would not cause out-of-bounds cursor movement – the cursor should be moved left or right, respectively[2]. Below are examples.

```
$ ./editor example_input_files/input2.txt
     *
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: d

      *
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: a

     *
```

[2]We’re going with WASD for cursor movement here. Sorry if you prefer a DVORAK keyboard layout.

```
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: a

     *
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: d

      *
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: d

       *
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: d


... (I've omitted much of the output here) ...


                      *
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: d

                       *
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: d

                        *
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: d

                        *
     12345678901234567890
*  1|abcdef
   2|ab
   3|
   4|xyz
   5|Hi there
   6|Hi how are you
   7|I am good
   8|how are you
   9|blah
  10|blah blah blah blah
     12345678901234567890
Enter command: q

Goodbye!
```

5.6 Part #5 - Moving the Cursor Up/Down

Attempts at up and down cursor movement should be triggered by ‘w’ and ‘s’, respectively. The cursor cannot be moved to the nonexistent line #0, and it cannot be moved past the last line of the current buffer contents.

For the rest of this part’s directions, we talk about when *scrolling* occurs. Scrolling refers to the movement of the window that lets us see into the current buffer contents. The following rules dictate how scrolling up and down should work. I based these rules off of how mouse scrolling works in Sublime Text.

- **Scrolling up**: **Scrolling up can only occur when the user tries to move the cursor up while the cursor is at the top of the window.** Moreover, if the cursor is already at the first line, then scrolling cannot occur; otherwise, scrolling should proceed, with the cursor staying at the top of the window.
- **Scrolling down.** Let’s break this down into three scenarios:
  1. If the cursor is *not* at the last line but *is* at the bottom of the window, then scroll down, keeping the cursor at the bottom of the window (which means that the cursor advances one line).
  2. If the cursor is at the last line but *not* at the top of the window, then scroll down but keep the cursor on the same line of the current buffer contents.

3. If the cursor is at the last line of the file *and* at the top of the window, then scrolling down should be prohibited. In other words, although the user can scroll past the end of the file, they cannot cause the last line of the file to completely disappear.

Below are examples of how your program should behave after you have completed this part.

```
$ ./editor example_input_files/lines.txt
     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
     12345678901234567890
Enter command: s

     *
     12345678901234567890
   1|Line 1
*  2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
     12345678901234567890
Enter command: w

     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
     12345678901234567890
Enter command: w

     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
     12345678901234567890
Enter command: s

     *
     12345678901234567890
   1|Line 1
*  2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
     12345678901234567890
Enter command: s

     *
     12345678901234567890
   1|Line 1
   2|Line 2
*  3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
     12345678901234567890
Enter command: s


... (I've omitted much of the output here) ...


     *
     12345678901234567890
   1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
*  8|Line 8
   9|Line 9
  10|Line 10
     12345678901234567890
Enter command: s

     *
     12345678901234567890
   1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
*  9|Line 9
  10|Line 10
     12345678901234567890
Enter command: s

     *
     12345678901234567890
   1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
* 10|Line 10
     12345678901234567890
Enter command: s

     *
     12345678901234567890
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
* 11|Line 11
      12345678901234567890
Enter command: s

      *
      12345678901234567890
   3|Line 3
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
  11|Line 11
* 12|Line 12
      12345678901234567890
Enter command: s

      *
      12345678901234567890
   4|Line 4
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
  11|Line 11
* 12|Line 12
  13
      12345678901234567890
Enter command: s

      *
      12345678901234567890
   5|Line 5
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
  11|Line 11
* 12|Line 12
  13
  14
      12345678901234567890
Enter command: s

      *
      12345678901234567890
   6|Line 6
   7|Line 7
   8|Line 8
   9|Line 9
  10|Line 10
  11|Line 11
* 12|Line 12
  13
  14
  15
      12345678901234567890
Enter command: s


```

```
... (I've omitted much of the output here) ...


     *
     12345678901234567890
   9|Line 9
  10|Line 10
  11|Line 11
* 12|Line 12
  13
  14
  15
  16
  17
  18
     12345678901234567890
Enter command: s

     *
     12345678901234567890
  10|Line 10
  11|Line 11
* 12|Line 12
  13
  14
  15
  16
  17
  18
  19
     12345678901234567890
Enter command: s

     *
     12345678901234567890
  11|Line 11
* 12|Line 12
  13
  14
  15
  16
  17
  18
  19
  20
     12345678901234567890
Enter command: s

     *
     12345678901234567890
* 12|Line 12
  13
  14
  15
  16
  17
  18
  19
  20
  21
     12345678901234567890
Enter command: s

     *
     12345678901234567890
* 12|Line 12
  13
  14
  15
  16
  17
  18
  19
  20
  21
     12345678901234567890
Enter command: w

     *
     12345678901234567890
* 11|Line 11
  12|Line 12
  13
  14
  15
  16
  17
  18
  19
  20
     12345678901234567890
Enter command: w

     *
     12345678901234567890
* 10|Line 10
  11|Line 11
  12|Line 12
  13
  14
  15
  16
  17
  18
  19
     12345678901234567890
Enter command: s

     *
     12345678901234567890
  10|Line 10
* 11|Line 11
  12|Line 12
  13
  14
  15
  16
  17
  18
  19
     12345678901234567890
Enter command: s

     *
     12345678901234567890
  10|Line 10
  11|Line 11
* 12|Line 12
  13
  14
  15
  16
  17
  18
  19
     12345678901234567890
Enter command: s

     *
     12345678901234567890
  11|Line 11
* 12|Line 12
  13
  14
  15
  16
  17
  18
  19
  20
     12345678901234567890
Enter command: s

     *
     12345678901234567890
* 12|Line 12
  13
  14
  15
  16
  17
  18
  19
  20
  21
     12345678901234567890
Enter command: w

     *
     12345678901234567890
* 11|Line 11
  12|Line 12
  13
  14
  15
  16
  17
  18
  19
  20
     12345678901234567890
Enter command: w

     *
     12345678901234567890
* 10|Line 10
  11|Line 11
  12|Line 12
  13
  14
  15
  16
  17
  18
  19
     12345678901234567890
Enter command: w

     *
     12345678901234567890
*  9|Line 9
  10|Line 10
  11|Line 11
  12|Line 12
  13
  14
  15
  16
  17
  18
     12345678901234567890
Enter command: q

Goodbye!
```

5.7 Part #6 - Moving the Cursor Repeatedly

Moving the cursor one spot at a time is a nightmare, so in this part, you will allow an integer to be given after any of ‘a’, ‘d’, ‘w’, or ‘s’ to indicate the number of times to perform that cursor movement. For instance “a 5” is the same as entering ‘a’ five times. Below are examples.

```
$ ./editor example_input_files/small.txt
     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: d 3

        *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: a 2

      *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: s 3

      *
     12345678901234567890
   1|Line 1
   2|Line 2
   3|Line 3
*  4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: w 1

      *
     12345678901234567890
   1|Line 1
   2|Line 2
*  3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
```

```
Enter command: s 4

      *
     12345678901234567890
   3|Line 3
   4|Line 4
*  5|Line 5
   6
   7
   8
   9
  10
  11
  12
     12345678901234567890
Enter command: w 100

      *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: d 80

                        *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: q

Goodbye!
```

5.8 Part #7 - Repeating the Previous Command

If the user enters nothing, then your program should run the last command that the user entered. If this occurs for the first command (i.e. if there is no previous command), then an appropriate message should be printed, as shown below.

Below are examples.

```
$ ./editor example_input_files/small.txt
     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: d 2

       *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command:

         *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: s

         *
     12345678901234567890
   1|Line 1
*  2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command:

         *
     12345678901234567890
   1|Line 1
   2|Line 2
*  3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: q

Goodbye!
$ ./editor example_input_files/small.txt
     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
```

```
Enter command:

No previous command.

      *
      12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
      12345678901234567890
Enter command: q

Goodbye!
```

5.9 Part #8 - Insertion

To insert, the user can enter ‘i‘ followed by a space and the string to insert (which may consist of multiple words/whitespaces). Insertion should begin at the cursor’s position. Below is an example of a simple insertion scenario involving saving. After this example, I talk about more complicated scenarios.

```
$ ./editor example_input_files/input4.txt
      *
      12345678901234567890
*  1|blah blah blah
   2|blah blah blah
   3|blah blah blah
   4|blah blah blah
   5|blah blah blah
   6|blah blah blah
   7
   8
   9
  10
      12345678901234567890
Enter command: i ABCD

      *
      12345678901234567890
*  1|ABCD blah blah
   2|blah blah blah
   3|blah blah blah
   4|blah blah blah
   5|blah blah blah
   6|blah blah blah
   7
   8
   9
  10
      12345678901234567890
Enter command: d

       *
      12345678901234567890
*  1|ABCD blah blah
   2|blah blah blah
   3|blah blah blah
   4|blah blah blah
   5|blah blah blah
   6|blah blah blah
   7
   8
   9
  10
      12345678901234567890
Enter command: s 4
```

```

      *
     12345678901234567890
   1|ABCD blah blah
   2|blah blah blah
   3|blah blah blah
   4|blah blah blah
*  5|blah blah blah
   6|blah blah blah
   7
   8
   9
  10
     12345678901234567890
Enter command: i AA $$ BB

      *
     12345678901234567890
   1|ABCD blah blah
   2|blah blah blah
   3|blah blah blah
   4|blah blah blah
*  5|bAA $$ BB blah
   6|blah blah blah
   7
   8
   9
  10
     12345678901234567890
Enter command: save output

      *
     12345678901234567890
   1|ABCD blah blah
   2|blah blah blah
   3|blah blah blah
   4|blah blah blah
*  5|bAA $$ BB blah
   6|blah blah blah
   7
   8
   9
  10
     12345678901234567890
Enter command: q

Goodbye!
$ cat output
ABCD blah blah
blah blah blah
blah blah blah
blah blah blah
bAA $$ BB blah
blah blah blah
$ diff example_input_files/input4.txt output
0a1
> ABCD blah blah
4,5c5
< blah blah blah
< blah blah blah
---
> bAA $$ BB blah
```

As you've seen, not every line in the file will be 20 characters wide, but the cursor can reach the rightmost column regardless. If the user attempts to insert past the end of a line in a file, then your editor should extend that line. Below is an example.

```
$ wc -m example_input_files/tiny.txt
4 example_input_files/tiny.txt
$ cat example_input_files/tiny.txt
ABC
$ ./editor example_input_files/tiny.txt
     *
     12345678901234567890
```

```
*  1|ABC
   2
   3
   4
   5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: d 5

          *
     12345678901234567890
*  1|ABC
   2
   3
   4
   5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: i past the end

          *
     12345678901234567890
*  1|ABC  past the end
   2
   3
   4
   5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: save less_tiny.txt

          *
     12345678901234567890
*  1|ABC  past the end
   2
   3
   4
   5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: q

Goodbye!
$ cat less_tiny.txt
ABC  past the end
```

If the insertion would go past column #20, then insertion should continue on the next line. If the next line does not already exist, then you should add a new line and continue from there. If a new line cannot be added (because it would be line #31), then insertion should not continue. Below are examples.

```
 ./editor example_input_files/input1.txt
     *
     12345678901234567890
*  1|abcdef
   2|ghi
   3|jklm
   4|jkj
   5|jklasj
   6|jklajs;lkfj
   7|qiowuioj
   8|lkjw;lkj
   9|qwklejklj;c
  10|jkaljsoiu
     12345678901234567890
Enter command: d 17

                      *
     12345678901234567890
*  1|abcdef
   2|ghi
   3|jklm
   4|jkj
   5|jklasj
   6|jklajs;lkfj
   7|qiowuioj
   8|lkjw;lkj
   9|qwklejklj;c
  10|jkaljsoiu
     12345678901234567890
Enter command: s 3

                      *
     12345678901234567890
   1|abcdef
   2|ghi
   3|jklm
*  4|jkj
   5|jklasj
   6|jklajs;lkfj
   7|qiowuioj
   8|lkjw;lkj
   9|qwklejklj;c
  10|jkaljsoiu
     12345678901234567890
Enter command: i ABCDEFGHIJKLMNOPQRSTUVWXYZ

                      *
     12345678901234567890
   1|abcdef
   2|ghi
   3|jklm
*  4|jkj              ABC
   5|DEFGHIJKLMNOPQRSTUVW
   6|XYZajs;lkfj
   7|qiowuioj
   8|lkjw;lkj
   9|qwklejklj;c
  10|jkaljsoiu
     12345678901234567890
Enter command: s 100

                      *
     12345678901234567890
* 27|keljweiourpiu
  28
  29
  30
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: i add a new line

                      *
     12345678901234567890
* 27|keljweiourpiu    add
  28| a new line
  29
  30
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: s

                      *
     12345678901234567890
  27|keljweiourpiu    add
* 28| a new line
  29
  30
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: d 2

                        *
     12345678901234567890
  27|keljweiourpiu    add
* 28| a new line
  29
  30
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: i hi

                        *
     12345678901234567890
  27|keljweiourpiu    add
* 28| a new line        h
  29|i
  30
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: s

                        *
     12345678901234567890
  27|keljweiourpiu    add
  28| a new line        h
* 29|i
  30
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: i hi

                        *
     12345678901234567890
  27|keljweiourpiu    add
  28| a new line        h
* 29|i                  h
```

```
  30|i
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: s

                        *
     12345678901234567890
  27|keljweiourpiu    add
  28| a new line        h
  29|i                  h
* 30|i
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: a 2

                      *
     12345678901234567890
  27|keljweiourpiu    add
  28| a new line        h
  29|i                  h
* 30|i
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: i Writing this whole thing will not work because this horrible editor does not let you go past line 30.

                      *
     12345678901234567890
  27|keljweiourpiu    add
  28| a new line        h
  29|i                  h
* 30|i                Wri
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: save output.txt

                      *
     12345678901234567890
  27|keljweiourpiu    add
  28| a new line        h
  29|i                  h
* 30|i                Wri
  31
  32
  33
  34
  35
  36
     12345678901234567890
Enter command: q

Goodbye!
$ diff example_input_files/input1.txt output.txt
4,6c4,6
```

```
< jkj
< jklasj
< jklajs;lkfj
---
> jkj               ABC
> DEFGHIJKLMNOPQRSTUVW
> XYZajs;lkfj
27c27,30
< keljweiourpiu
---
> keljweiourpiu    add
>  a new line        h
> i                  h
> i                Wri
$ wc -l example_input_files/input1.txt
27 example_input_files/input1.txt
$ wc -l output.txt
30 output.txt
```

5.10 Part #9 - Edit History

In this part, you will modify the editor so that *insertions* (no other operations) can be undone or redone by entering 'u' or 'r', respectively. This requires that your program maintain undo and redo histories.

When I say "new insertion" in this paragraph, I am referring to an insertion that is not caused by an undo operation or a redo operation. The undo operation should undo the last insertion done, whether that insertion was a "new insertion" or the result of a redo operation. The redo operation should redo the last insertion that was undone. If the user performs a "new insertion", then the redo history should be erased.

Note that undo/redo operations do not care where the cursor or window are and do not change where the cursor or window are.

You are required to use a stack (or two stacks) to do this part. You should use `std::stack` from the STL.

v.3 Update: You may assume that the autograder will never do an undo operation that causes the cursor to go out of bounds (by undoing the insertion/creation of the line that the cursor happens to be on).

Below are examples of how your program should behave. Notice the messages that should be printed if there are no actions to undo or redo.

```
$ ./editor example_input_files/small.txt
      *
      12345678901234567890
*   1|Line 1
    2|Line 2
    3|Line 3
    4|Line 4
    5|Line 5
    6
    7
    8
    9
   10
      12345678901234567890
Enter command: i Hello there

      *
      12345678901234567890
*   1|Hello there
    2|Line 2
    3|Line 3
    4|Line 4
    5|Line 5
    6
    7
    8
    9
   10
      12345678901234567890
Enter command: u

      *
      12345678901234567890
*   1|Line 1
    2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: u

Cannot undo.

     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: r

     *
     12345678901234567890
*  1|Hello there
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: r

Cannot redo.

     *
     12345678901234567890
*  1|Hello there
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: i ABC

     *
     12345678901234567890
*  1|ABClo there
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: s
```

```

     *
     12345678901234567890
   1|ABClo there
*  2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: i DEF

     *
     12345678901234567890
   1|ABClo there
*  2|DEFe 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: s

     *
     12345678901234567890
   1|ABClo there
   2|DEFe 2
*  3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: i GHI

     *
     12345678901234567890
   1|ABClo there
   2|DEFe 2
*  3|GHIe 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: r

Cannot redo.

     *
     12345678901234567890
   1|ABClo there
   2|DEFe 2
*  3|GHIe 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: u

     *
     12345678901234567890
   1|ABClo there
   2|DEFe 2
*  3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: u

     *
     12345678901234567890
   1|ABClo there
   2|Line 2
*  3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: u

     *
     12345678901234567890
   1|Hello there
   2|Line 2
*  3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: r

     *
     12345678901234567890
   1|ABClo there
   2|Line 2
*  3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: i XYZ

     *
     12345678901234567890
   1|ABClo there
   2|Line 2
*  3|XYZe 3
   4|Line 4
   5|Line 5
   6
   7
   8
  9
 10
     12345678901234567890
Enter command: r

Cannot redo.

     *
     12345678901234567890
  1|ABClo there
  2|Line 2
*  3|XYZe 3
  4|Line 4
  5|Line 5
  6
  7
  8
  9
 10
     12345678901234567890
Enter command: u

     *
     12345678901234567890
  1|ABClo there
  2|Line 2
*  3|Line 3
  4|Line 4
  5|Line 5
  6
  7
  8
  9
 10
     12345678901234567890
Enter command: ... (I've omitted the rest of the output.) ...
```

5.11 Part #10 - Attempting to Quit with Unsaved Changes

If the user attempts to quit but has not yet saved the current state of the buffer contents to a file, then your program should inform the user of this and ask the user if he/she still wishes to quit, like what real editors (e.g. Sublime Text). Below are examples of the simple case. After these examples, I talk about why this is more complicated than it seems.

```
$ ./editor example_input_files/small.txt
     *
     12345678901234567890
*  1|Line 1
  2|Line 2
  3|Line 3
  4|Line 4
  5|Line 5
  6
  7
  8
  9
 10
     12345678901234567890
Enter command: i DEF

     *
     12345678901234567890
*  1|DEFe 1
  2|Line 2
  3|Line 3
  4|Line 4
  5|Line 5
  6
  7
  8
  9
 10
     12345678901234567890
```

```
Enter command: q

You have unsaved changes.
Are you sure you want to quit (y or n)?
n

     *
     12345678901234567890
*  1|DEFe 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: save blah

     *
     12345678901234567890
*  1|DEFe 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: q

Goodbye!
$ cat blah
DEFe 1
Line 2
Line 3
Line 4
Line 5
```

Once we consider the edit history, things become more complicated. On editors like Sublime Text, if – after a save operation – you do a modification and then perform an undo operation (or if you perform an undo operation followed by a redo operation), the editor will recognize that you are back to the state you last saved and not think that you have unsaved changes. However, if – again, after a save operation – you were to insert "DEF" to replace "ABC" and then insert "ABC" to replace the "DEF" you just inserted, these will be perceived as unsaved changes, even though the current state is identical to the last state saved. Your editor should emulate all of this behavior. Below are examples.

```
$ ./editor example_input_files/small.txt
     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: i ABC

     *
     12345678901234567890
*  1|ABCe 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: save output

     *
     12345678901234567890
*  1|ABCe 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: u

     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: q

You have unsaved changes.
Are you sure you want to quit (y or n)?
n

     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: r

     *
     12345678901234567890
*  1|ABCe 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: q

Goodbye!
```

```
$ ./editor example_input_files/small.txt
     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: i ABC

     *
     12345678901234567890
*  1|ABCe 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: i Lin

     *
     12345678901234567890
*  1|Line 1
   2|Line 2
   3|Line 3
   4|Line 4
   5|Line 5
   6
   7
   8
   9
  10
     12345678901234567890
Enter command: q

You have unsaved changes.
Are you sure you want to quit (y or n)?
y

Goodbye!
```

5.12 Epilogue: Additional Features

Do not add anything from this subsection into what you end up submitting to Gradescope.

Below is a non-exhaustive list of features you could add to this editor *on your own time* (again, not for your submission to Gradescope), in order to make it more complex and, perhaps, more résumé-worthy.

- Other forms of modification: deletion, find-and-replace, etc.
- Use the curses library to make the editor take up the entire terminal in one area, so that you don't just have to keep printing out the file's contents periodically. The curses library also allows you to react to inputs without the user having to press the Enter key. It's admittedly pretty difficult to explain what the curses library can do for you, because it can do so much, so I would recommend you go through a tutorial on it[3]; it really opens up the possibilities when it comes to terminal-based applications and would make your editor more usable by allowing you to not have to press the Enter key all the time.
- *Requires ECS 50 knowledge of bits and bit operators*: Support files that are encrypted in a certain way, e.g. with run-length encoding or Huffman encoding.

[3]It's possible that a later assignment in this course has you use the curses library; I'm not sure yet. (The problem is it's impossible to autograde such an assignment.)

- The ability to edit multiple files simultaneously (like having multiple tabs in an editor).
- A clipboard for copy/pasting.
- Whatever features make your editor so good that you can use it to write your code in future courses :-)

6 Grading Breakdown

As stated in the updated syllabus, this assignment is worth 9% of your final grade. Below is the breakdown of the worth of the test cases for each part (for a total of 90 points):

- Cases for part #1: 5 points. (Cases #1 through #5)
- Cases for part #2: 10 points. (Cases #6 through #7)
- Case for part #3: 5 points. (Case #8)
- Cases for part #4: 5 points. (Cases #9 through #10)
- Cases for part #5: 17.5 points. (Cases #11 through #17)
- Cases for part #6: 5 points. (Cases #18 through #19)
- Cases for part #7: 5 points. (Cases #20 through #21)
- Cases for part #8: 15 points. (Cases #22 through #27)
- Cases for part #9: 15 points. (Cases #28 through #31)
- Cases for part #10: 7.5 points. (Cases #32 through #34)

7 Autograder Details

Once the autograder is set up on Gradescope, I will send a Canvas announcement and add additional details below.

If you haven't already, you should read what I say in the syllabus about the Gradescope autograder.

As mentioned above, the *only* file that you should submit to Gradescope is `editor.cpp`. Any other file that you submit will be ignored. The autograder will compile your code with the below command.

```
g++ -std=c++14 -Wall -Werror editor.cpp -o editor
```

Your output must match mine *exactly*.

The autograder will grade your submission out of 90 points.

Even if you do not finish a specific part, you might still be able to pass some of the test cases corresponding to later parts. For instance, even if you cannot get saving to a file to work, you could still pass test cases concerning scrolling up/down and undo/redo (except for the ones that happen to involve saving).

Each test case corresponds to a shell script that you can find in the `case_scripts` folder on Canvas. Scripts in this folder often get their standard input from files that you can find in the `stdin` folder on Canvas. Of course, if your understanding of Linux and shell scripting is not that great, it might limit your ability to understand what is going on in the shell scripts, which is why I encouraged you to make yourself more comfortable with Linux and shell scripting in a Canvas announcement sent towards the start of this quarter.

At least one of the hidden test cases is failed if your code has memory leaks.

7.1 Regarding Timeouts

As this is a bigger assignment with a lot more output for the autograder to evaluate, it is possible that the autograder can "give up" if there is too much output, which usually happens if your program did not end, whether due to an infinite loop or because it did not terminate when expected. When this happens, you could see a general message about the autograder timing out, or you could see messages like the below on specific test cases.

```
Test Failed: Command '['bash', './run.sh']' timed out after 30 seconds
```

```
Test Failed: [Errno 12] Cannot allocate memory
```

If you get a case-specific error message, then you should run the visible test case's input on your end (on the CSIF) and debug from there. If your program is behaving properly on the CSIF, then feel free to email me.

If you get a general autograder timeout message, then you should run all visible test cases' inputs on your end and debug from there. If your program is behaving properly on the CSIF, then feel free to email me.

If the autograder is taking a long time, it is probably not a good idea to just keep resubmitting the same code over and over and hope that something different somehow happens.

If you get the below message, then please email me with the URL of this Gradescope submission. In the meantime, you should run the visible test cases' inputs on your end and see if they all work.

```
The autograder failed to execute correctly. Please ensure that your submission is valid. Contact your
    course staff for help in debugging this issue. Make sure to include a link to this page so that they
    can help you most effectively.
```

UCDAVIS
COMPUTER SCIENCE