The objectives of this project are (1) the design and implementation of a simple “editor” for lines of text; (2) giving you practice with a particular data structure: Sequences of strings implemented with dynamic doubly-linked lists of dynamically allocated C-strings; and (3) giving you further practice with solving programming problems that involve sequences (of characters) implemented by arrays.

For this project, the use of (the admittedly primitive) C-strings is REQUIRED (even if you know better ways.) In the projects after the midterm, you will have the option of using STL (Standard Template Library) classes for strings.

At this point, it would be very wise to read the specification below. After it, read the section on how to do the project. Then make sure you are familiar enough with the course contents featured in Main and Savitch’s Chapters 4 and 5 required for the data structure implementation. All the Chapter 5 operations or certain variants of them will be needed for this project, except for list copying. Study the pseudo-code and sample implementations as necessary. The essentials will be covered in the upcoming lectures. When you begin draft design, creating some C++ files (RCS use **required**) and the build script, and continue to complete the project. Keep the this assignment sheet handy so you can consult it for details, to write notes and a “to-do” list.

## 1 Specification of Commands, other inputs and actions

This editor maintains for you (the user) a main list of text lines. The editor also has a cursor that marks, conceptually, the front of any one of the lines or the end of the list. Thus, if the list has \( n \) lines, the cursor will be at one of \( n + 1 \) positions.

With various combinations of commands you can reposition the cursor, print the line after the cursor, insert a line where the cursor is, delete lines, etc.

The editor also maintains a cut-list which holds the lines that have been deleted, until the cut-list is cleared by various commands. One such command pastes the cut-list into the main list at the cursor position, and then clears the cut-list, which makes the cut-list empty.

Note that manual editing of the characters inside each line is not to be done: Except for one command to justify each line, everything the program should do only involves inputing, manipulation or discarding entire lines.

We will evaluate most of the functionality of your project work in the order of the description of the commands below.

In **Fixed Font Type**: Each command or a general syntactic description of it is given first. Under some commands is given the exact text(s) or form of messages your program must print to report some specified result or when certain user error conditions occur.

Each command and message(s) are followed by an explanation, printed in Roman Font Type. Each command and message text will soon be available in the file below on itsunix

```
~acsi310/Proj2/strings.txt
```

so you can copy and paste them into your program instead of typing them. **For full credit, messages must be printed exactly as shown.**
Unlike those in a practical “user manual” type specifications, the explanations include information about how each command should be implemented because learning these implementation techniques is a project objective.

First implementation technique item: There must be a separate “user interface” module for inputting commands and line input. It is separate from a “core” editor module (to be implemented by a C++ class named “editorCore”) that maintains and manipulates the list, cursor and cut-list.

For the user interface module: Make use of the command line study and pattern material that appeared in Lecture 3, which uses 
`strcmp` (covered in Ch. 4). Write new code to read one data input line in response to the :input1 command.

Then add special code for the cases when data, such as a line number, will follow a command.

### 1.1 Commands and other input operations one by one

1. **:⟨misspelled, empty or unimplemented command⟩**
   
   ERR: I don’t know that command.

   A **command** is any input line whose first character is a colon.

   If the characters following the colon don’t follow the syntax of the commands defined below that you implemented, print the indicated error message and continue to input and process lines, except when the last input caused the program to exit.

2. **:quit**

   The quit command makes the editor exit. Errors are impossible!

3. **:about**

   Print 2-10 lines about this program, this assignment and yourself, including your name and anything else you want to.

   You may make your program print a short menu or reminder about its commands in the message printed here.

   NOTE: Unlike project 1, the program MUST NOT print a prompt or a menu before or after each command or input.

4. **:input1**

   Make the program accept one line of input to be used as data, not a command.

   That line will then be inserted into the text being edited *at the cursor position*.

   If the user types a line that is too long, the program should print


   and then exit.

   When a line is inserted, it is inserted at the position of the cursor and the cursor is then positioned **JUST AFTER THE NEW LINE!!** That way, successively typed input lines will go into the main list in the order they were typed in.
All terminal input must be read using the following code: (For simplicity, don’t bother removing any trailing whitespace from commands or text input lines.)

```cpp
#include <iostream>
#include <cstdlib> //supplies exit()
using package std;
...
const int INBUFSIZE = 72;
char inbuf[INBUFSIZE];

if( !cin.getline(inbuf, INBUFSIZE) )
{
    cout << "ERR:Line input failed(too long?end of file?)..exiting." << endl;
    exit( 0 );
}
```

The call to `cin.getline()` reads an input line up to `INBUFSIZE` characters long **including** the terminating newline character. The line includes any leading or trailing **whitespace** characters (spaces or tabs here). The `cin.getline()` call fails if the input is too long. This call stores the inputted characters into the given array except that it replaces the terminating newline with the NULL `char` value. That replacement turns the inputted characters into a **C-string**. Ask yourself and become sure of the answer: What is the maximum length of a line this software will handle?

These rules apply even to the **empty** input line. Users will type that in by pressing the enter key only. It’s important to understand the empty C-string: It’s a NON-EMPTY array of the one character ‘\0’.

The length of the useful data in each line varies, up to a maximum of 72 characters. An data input line can be empty, since the user may want to put a blank line in his/her text. Whether it be empty or not, the program must determine its length and **add 1 to the length** in order to determine how long an array of `char` to dynamically allocate. (Why? What can happen if 1 isn’t added?) After dynamically allocating (using `new[]` of course..) the array, the line must be copied into the array. This array will become part of your `editorCore` data structure. The input buffer `inbuf` above is NOT part of `editorCore`’s data structure!

5. :prinall

Prints all the lines, one per output line. An empty line would appear blank. If the list (of lines) is empty, nothing is printed.

Thus, if you just type some non-command lines and then type

:prinall

you will see a copy of what you typed in, unless some of the non-command lines were too long.

The cursor position is UNCHANGED after :prinall.
6. :up
   ERR: You’re at the top.
   Move the cursor up one line. Do nothing but print the indicated error message if this is impossible.
   Watch out that the cursor really moves. After an :up command, try inputting a new line and observe using :printall where it was inserted (it shouldn’t be at the bottom.)

7. :print
   ERR: You’re at the bottom.
   Print the current line, i.e., the line the cursor is in front of. In the one case when this is impossible because the cursor is at the end of the list, print the indicated error message.
   The following sequence reprints <non-command line>:
   
   :input1
   <non-command line>
   :up
   :print
   <non-command line>

8. :input1
   (There is an empty line above which you cannot see.)
   Make sure that if the user presses just the enter key, an empty line is inserted just like a non-empty line.

9. :where
   AT LINE NUMBER <number>
   <number> signifies the ordinal number of the line the cursor is in front of (1 for the first line, n for the last line if there are n lines, etc.). If the cursor is at the end, <number> signifies the number of lines plus 1, which would be the line number of a new line if a new line were appended at the end. So, if the line list is empty, <number> signifies 1.

10. :down
    ERR: You’re at the bottom.
    It’s the opposite of :up.

11. :cut
    ERR: You’re at the bottom.
    The current line is removed from the list, except if the cursor is at the end of the list. The cursor remains in place (in front of the next line, if any, or at the end if not).
    The editor maintains a second list of lines called the cut-list. Each time a :cut command is successful, the line removed is appended to the end of the cut-list.
12. **:printcuts**
   Print all the lines in the cut-list in order. Print nothing if the cut-list is empty.

13. **:clearcuts**
   ERR: Cut list is empty.
   Make the cut-list empty. Print the error message if it is already empty.

14. **:paste**
   Insert the lines from the cut-list into the main list at the position of the cursor. Do nothing
   if the cut-list is empty.
   Leave the cursor just after the newly inserted lines.
   Make the cut-list empty.
   (What do the rules about cut, clearcuts and paste imply about when editorCore should run
   delete[] to recycle storage used for the char arrays?)

15. **:go <line number>**
   ERR: Line number <line number> doesn’t exist.
   (<line number> is a placeholder that signifies a number expressed in decimal. This notation
   is common for describing the syntax of commands and programming language statements.)
   Move the cursor to the given numbered line. Print the error message and leave the cursor
   unmoved if there are fewer than <line number>-1 lines in the list.
   (The user interface will have to check and process the characters after the :, g, o and space
   as decimal digits. It then needs to figure out the number.
   Strategies for solving this problem will be discussed in lectures and lab session.

16. **:justifyall 72**
   This commands the editor to process every line in the main list to make it into a line so that
   (a) Each line is exactly 72 characters long.
   (b) Except for an empty line or a line with just one word, each line both begins and ends
   with a non-blank character.
   (c) Except for an empty line or a line with just one word, the number of blanks between
   each adjacent pair of words is as equal as possible.
   In some situations, you will have the same number of blanks between each pair of adjacent words.
   In the other situations, some adjacent word pairs will be separated by n blanks and the
   other adjacent word pairs will be separated by n + 1 blanks, for some number n. In these
   situations, the spaces of n blanks should all be first on the justified line, and after all of
   them, the spaces should be n + 1 blanks.
   In this specification, a **word** means a maximal substring of non-blank characters.
   For example if the original line is
Here is an example, a bunch...of Silly’s Stuff. Extra 9 er*
123456789012345678901234567890123456789012345678901234567890123456789012

justified line should be

Here is an example, a bunch...of Silly’s Stuff. Extra 9 er*
123456789012345678901234567890123456789012345678901234567890123456789012

(I printed rows of 72 digits so you can see the positions of each character in the above lines.)

Optional: Instead of only justifying lines into a length of 72, make your program accept any number between 1 and 72 after the :justifyall command, and justify to the line length given to that number. Your program may either truncate lines that are too long, justify them with one space between words and let them exceed the given length, or, (if you are really ambitious) break them the way a real word processor would do.

We will not grade the optional functionality. It is allowed because debugging on shorter lines may be more pleasant; and to encourage more ambitious work.

The remaining commands are for honor’s points because they require learning about files which is not a core CSI310 topic.

17. :printall <filename>
ERR:Can’t open file.

Write all the lines into a file, overwriting anything in the file already there, exactly as if they were output by the plain printall command. Then, resume handling user input.

18. :infile <filename>
ERR:Can’t open file.

Read and process lines from a file exactly as if its contents were typed in, then resume handling user input. (Thus, instead of exiting on end-of-file, resume input from cin.)

This requires tricky replacement of the simple input error checking code above. If myfile is the input file stream, then myfile.getline(inbuf, INBUFSIZE) evaluates to false in two different situations: (1) The current line is too long. (2) The end of file has been reached. In the former case, myfile.gcount() returns the (non-zero) number of characters actually read.

(You can assume all lines in our test input files will end with the newline character.)

2 How to Do This Project; Some Graded Items!

For design practice, you must design a class named class editorCore whose public member functions provide the core operations on two lists of C-strings. This class must implement two sequences of strings, one for the our list and the other for the cut-list, using doubly linked lists of nodes, where each node holds a pointer to a char array that holds a C-string. The implementation’s data structure consists of private data members of editorCore and dynamically allocated linked list nodes.
that hold pointers to `char` arrays. The private data members must of course include head and tail pointers to the linked lists. They must include something to implement the cursor. You might choose to include other private data members for such things as the current line number, etc.

It is up to you how much of the rest of the program is designed and implemented using classes and other object oriented practices. One choice is to make use of Main and Savitch’s “linked list toolkit”. Another (my preference for CS2 at Albany) is to use a `struct` for list nodes and let `editorCore`'s member functions manipulate node fields directly.

For simplicity, I think it would be a good idea to provide public functions of class `editorCore` to print using `cout` (or an argument output stream if you get to implementing `printall` to a file).

It is your design choice about whether `editorCore`'s functions detect and/or print messages about user errors described below, or whether the `editorCore` should enforce preconditions and its caller detect and act on user errors. It is smart to think about what would happen with user errors before you commit much time into the design of `editorCore`.

The job of (an instance of) class `editorCore` is to hold the data being edited, carry some out changes to it, and print (at your design option) copies of selected contents. Following this design requirement, class `editorCore` MUST NOT obtain input from `cin` or a file, NOR figure out commands by examining input strings! If the user interface language changes, the `editorCore` class must be reusable without change.

One critical design choice for `editorCore` is “Will `editorCore` call `new` to allocate space for a line and then copy the characters given to it, or (alternatively) will the caller (probably the user interface module) call `new` and pass to `editorCore` the address of the new string?” In either case, it’s `editorCore`’s job to run `delete` on this address after the line is removed as a result of the `clearcuts` operation.

Part of the project grade will come from your writing of the `editorCore` class's header file `editorCore.h` to document its design expressed using pre and post condition definitions of its public functions. In addition to this documentation of the public function members, comments to document the data members (where the data structure begins), preferably written in the form of invariants, will also be graded.

I suggest you write the user interface in a “main” module much like the test driver of the previous project and Main/Savitch’s sample programs. This main module, or perhaps classes of your own design if you choose that route, but not of class `editorCore`, should analyze the user input commands! One main difference is that the user interface module should read in whole lines of input, not just single characters. It could use C-string library function `strcmp()`, described in DSO Chapter 4, to test if the string given by the user equals a legal command (see below).

1. All the software development practices given on the project 1 assignment must also be followed for project 2 in order to earn full credit. Some details and additional requirements are given below.

2. Submission of the RCS revision history database for non-trivial files, showing your development process, is REQUIRED. It was ok (with points off) to omit it or start it late for the first project, but now, PROJECT 2 (and the rest) will earn 0 points WITHOUT IT. Right now is the time to learn basic RCS use, if you haven’t already.

3. Organization of source files must reflect the software’s design, or points will be taken off. In particular, at the very least, `editorCore.h`, `editorCore.cxx` and the `.cxx` file containing
the user interface code must be separate files. The file `editorCore.h` must document your `editorCore` class with pre and postconditions for all public functions.

4. An executable “build” script named `build.sh` is required for full credit. When run, the build script should compile and link the software to create an executable file named “`proj2`” in the same directory.

At this point, it would be wise to also write and test a “cleanup” script that automatically deletes any “core”, object or executable program files so that you can use it before submitting the project.

5. Submit a single directory to project name `Proj2` using “`turnin-csi310 -c csi310 -p Proj2 Directory`”. Verify with “`turnin-csi310 -v -c csi310 -p Proj2`”. As before, points will be deducted if object, “core” or executable files are submitted.

Acknowledgements

After reading some on-line documentation of `elvis`, the GNU variant of the classic Unix text editor `vi`, I based the user’s view design here loosely on the `ex` subset of `vi` commands.

The `ex` line editor was popular and well-suited on “teletype” style computer terminals. A teletype is like an electric typewriter that can also type the computer output by itself. They were popular in the days, long before PCs, when video terminals were too expensive for common use.

Simplified student implementation of “visual” style editors such as `emacs`, `pico`, `vi`, Microsoft’s `notepad`, text input boxes in Web pages and application software, etc. are a CSI402 level topic...