The objectives of this project are (1) the design and implementation of a simple “editor” for lines of text, and (2) giving you practice with a particular data structure: Sequences of strings implemented with dynamic doubly-linked lists of dynamically allocated C-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. (Maybe also use it 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 the 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 editing of the characters inside each line is not to be done: The only things this project should do is manipulate entire lines, and do simple searches within 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 is 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 information 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: Program with `strcmp` (covered in Ch. 4) to distinguish cases 1-4. Then add special code for case 5. `strcmp` will continue to work up to 15. Coding cases 16-19 requires understanding the array nature of C-strings.

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.

4. :help
   
   First, print 1-3 lines that explain how this editor is “line-based”.

   Then, for each command you have implemented, and for the line input operation, print 1-3 lines to tell people how to use the feature.

   (You may implement this with a file if you know how, so long as you submit the file in the same directory where the executable program file `proj2` will be built.)

5. `<A line that doesn’t begin with a : will self-insert at the cursor.`
   
   ERR: Line input failed(too long? end of file?)... exiting.

   The notation above signifies a line that doesn’t begin with a colon.

   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 non-command 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.)

   ```
   #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'.

Sometime after reading an input line, the program must check if the user entered the empty line, a line beginning with a colon or something else. To code this decision you will have to understand what getline() does.

If it is the empty line or a non-empty line not beginning with a colon, 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!

6. :printall

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 :printall
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 :printall.

7. :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.)
8. **: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>`:

   `<non-command line>`
   :up
   :print

9. **<An empty line will self-insert too.>**

   Make sure that if the user presses just the enter key, an empty line is inserted just like a non-empty line. (The above <> syntax symbol signifies the empty line.)

10. **: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.

11. **:down**
    
    ERR: You’re at the bottom.

    It’s the opposite of :up.

12. **: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.

13. **:showcuts**

    Print all the lines in the cut-list in order. Print nothing if the cut-list is empty.

14. **:clearcuts**

    ERR: Cut list is empty.

    Make the cut-list empty. Print the error message if it is already empty.

15. **: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?)

16. :<line number>go
ERR:Line number <line number> doesn’t exist.
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 detect if the char after the : is a decimal digit. It then needs to figure out the number and find the end of the <line number> substring. Here is how you can convert the char code for a digit into a number you can add or multiply:

```cpp
char ch;
int num;
....
num = ch - '0';
assert( (0 <= num) && (num <= 9));
```

Here’s how to convert the string "239" to the corresponding number after interpreting it in base ten:

```cpp
int num = 0;
num = 10*num + ('2' - '0');
num = 10*num + ('3' - '0');
num = 10*num + ('9' - '0');
assert( num == 239 );
```

Generalize this algorithm and code it in C++!

17. :<pattern string>/go
ERR:Line input failed(too long?end of file?)..exiting.
ERR:Unmatched /.
ERR:Empty pattern //.
ERR:Missing go after pattern.
ERR:There is no copy of that pattern at or after current line.
<pattern string> is an non-empty string not containing the slash character ’/’. (This command is subject to the same input length constraint given before.)
If there are no other errors, position the cursor in front of the first line at or after the cursor in which <pattern string> appears as a (contiguous) substring.
Searching each string for a pattern substring is easy to program by calling strstr(), a standard C library string function whose documentation is available on our systems, the Internet, various books, or the staff if you have trouble finding it.
You will have to write your own code to scan the input buffer to calculate a pointer to the beginning of <pattern string>, locate the end of it, and overwrite the terminating ’/’ with a null character ’\0’ to produce a C-string suitable for strstr().
18. `: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.

19. `: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, `myinfile.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 `class 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 \texttt{new} and pass to \texttt{editorCore} the address of the new string?” In either case, it’s \texttt{editorCore}’s job to run \texttt{delete} on this address after the line is removed as a result of the \texttt{:clearcuts} operation.

Part of the project grade will come from your writing of the \texttt{editorCore} class’s header file \texttt{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 \texttt{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 \texttt{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, \textbf{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, \texttt{editorCore.h}, \texttt{editorCore.cxx} and the \texttt{.cxx} file containing the user interface code must be separate files. The file \texttt{editorCore.h} must document your \texttt{editorCore} class with pre and postconditions for all public functions.

4. An executable “build” script named \texttt{build.sh} is required for full credit. When run, the build script should compile and link the software to create an executable file named “\texttt{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 \texttt{proj2} using

   “\texttt{turnin-csi310 -c csi310 -p proj2 Directory ”}. Verify with “\texttt{turnin-csi310 -v -c csi310 -p proj2”}.

   As before, points will be deducted if object, “core” or executable files are submitted.

\textbf{Acknowledgements}

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