Programming Plan

Textedit

**Description:**

A simple, console based text editor with nano style cursor control. The cursor should be movable using the four arrow keys, the PgUp and PgDown keys, the Home and End keys, and by the search function. Files should be able to be saved and loaded from within the program, and opened by passing them as arguments on the command line. Users should be able to save files to a different filename if they wish. Users should be able to highlight using shift+arrow keys, copy using ctrl-g, and paste using ctrl-v. Users should be able to search for text using ctrl-r. The program should implement error handling for violations of file permissions, attempts to save with no hard-drive space, and attempts to save to an invalid file name.

**Components:**

* Text rendering object: Reads text from the buffer, displays it on the screen, displays hotkey reference, displays error messages.
* Buffer: Stores text. A basic vector or string object would likely suffice.
* User input object: Reads input from the user, catches hotkey presses.
* File-system object: Loads and unloads files from the buffer, possible autosave?
* Search object: can likely just use string::find here.

**Intended Users:**

* Anyone who needs to edit simple, unformatted text files.

**Appeal:** (Why will the program be of use for the intended users?)

* Offers search function for text.
* Allors editing in-place and insertion of text anywhere in the file.
* Allows copy-pasting.

**Data Input:**

* Console window size.
* Text input.
* Hotkey presses.
* File names.

**Advanced Concepts:**

* The current project I have in visual studio for this program has multiple classes with defined constructors and destructors.
* The error-handling class uses inheritance to support handling different errors in different classes.
* The text-buffer class I have could be considered an abstract-data-type, as it presents a continuous string of text, but operates in terms of a vector of lines so it can work with the text-rendering class.
* The error-handling class provides for handling of logic exceptions, but currently doesn’t work for internal exceptions thrown by libraries. Will work to expand it & add in try-catch blocks in misbehaving areas.
* I’ve overridden assignment operators for the text-buffer class to make writing text to it easier.

**Algorithm:**

Currently, my program has two major algorithms. The input tracker loop, and the text-render loop. I’ll explain how both work in separate paragraphs below, along with a third paragraph about the menu function I wrote as it’s probably the most complicated function in my program:

* **Input Tracker**: The input tracker class consists of two functions, the constructor, which spawns a thread, and the thread itself. This thread contains a loop which reads events from the keyboard and dispatches them to various other classes. This loop contains a check for a mutex to allow the main thread to control the keyboard if necessary. The logic inside the loop works as follows:
  + Loop checks lock state of mutex. If locked, it sleeps for 10ms and continues to the next iteration
  + Loop checks if windows has events buffered using API call. If not, it sleeps for 10 ms and continues to the next iteration.
  + Loop locks mutex.
  + Loop reads input records into array using API call.
  + Loop iterates through array, saving KEY\_EVENT’s to a vector and discarding other events.
  + Loop sends vector to dispatch function.
    - Dispatch function iterates through array of events and checks if a derived class of key\_press\_handler is registered to handle the specific key event.
    - If so, it calls process\_keypress function provided by the derived class.
      * Depending on which derived class is called, this keypress could either be sent to text\_buffer, for letters, numbers, or symbols, menu or text\_render, for arrow keys, depending on the state of the program, or dispatch, for CTRL-Q.
  + Loop frees array and clears the vector.
  + Loop unlocks the mutex.
  + Loop sleeps for 10 ms and repeats.
* **Text Render:** Text render is the most important class in my program. It handles displaying the text from the buffer onto the screen and allowing for arrow-key control, along with inplace editing. Right now the class is still in the works, but I have a broad plan for it.
  + Text Buffer passes vector containing contents to Text Render
  + Text Render takes contents, gets screensize, and determines if anything would wrap.
  + If so, it replaces the last character that could display without wrapping with an ampersand and registers the string as wrapping for later.
  + Text Render waits for user input and rereads buffer whenever a change is registered, redisplaying to the screen once every 20 ms.
  + If arrow keys are pressed, text buffer gets a handle to standard output and checks if the arrow key would move the cursor off screen.
    - If so, it either scrolls the buffer or creates a substring of the wrapping string that can fit on the screen. As an example, if the screen size is 120 chars and we have a 240 char string, that string would be cut to [0, 120). If the cursor was at 119 and pressed right, the new string would be [1, 121).
    - If not, text render gets the current cursor position using the Windows API and shifts it by the value indicated by the keypress.
  + If the backspace key is pressed, Text Render gets the position of the user cursor, finds text one to the right of the cursor, and removes it from the respective vector, sending a line number and offset to text-buffer.
  + Text Render cuts off the last 2 lines of the console and places a string containing program hotkeys there.
* **Menu:**
  + Menu is called with 4 arguments, 2 strings, menu\_name and menu\_cursor, and a vector of strings, items.
  + Menu’s logic starts with a do-while loop. This loop runs until the user presses the enter key.
  + The first section of the loop consists of a switch statement used to modify the chosen\_opt variable. This statement switches on the variable arrow\_old, which starts out set to zero, but is set based on the user pressing the arrow key later in the program. Up and right presses decrement the variable chosen\_opt, down and left increment chosen\_opt.
  + Chosen\_opt is limited to a range of [1, items.size()] using min and max.
  + A string, format, is created from “[\*]” menu\_name, and “:” in that order.
  + A for loop is used to iterate through the items variable:
    - First, A newline and 5 spaces are prefixed to format.
    - Next, if the chosen\_opt variable equals the loop counter, menu\_cursor is prefixed.
    - Then, one of the item variables is added to format.
    - Finally, the loop counter is incremented.
  + The program then displays format, resets the loop counter, and waits for the user to press either one of the arrow keys or the enter key.
  + If the user presses an arrow key, the loop repeats and chosen\_opt is modified.
  + If the user presses enter, the loop terminates, returning chosen\_opt.

**Error Handling:**

As my program is a text editor, the two exceptions I’ve added so far that I intend to keep for the final project just have to do with handling user input of bad file names. All other exceptions I’ve added in are used for debugging. In terms of the examples given in the directions, none of those seem to apply past file-name input. Bad user input is possible when inputting a file name, and I’ve guarded against that with a simple regex. The program also detects if the user attempts to open a file that doesn’t exist, and throws an exception so the user can choose the “Open File” option. If nothing is entered, the user either stays at the menu or the file stays in the same state it was in when opened. The denominators for all division statements in the program are either constant or secure against being zero. As an example, the following statement is used to calculate the y value of a point in an array, given that point’s index. “const int array\_y = floor(array\_pos / window.X)” As window.x cannot be zero according to Microsoft, “The function also fails if the Right member (or calculated right coordinate) is less than or equal to the Left member (or calculated left coordinate) or if the Bottom member (or calculated bottom coordinate) is less than or equal to the Top member (or calculated top coordinate),” a divide by zero cannot happen. If a user inputs lower case instead of upper case, they get an improperly capitalized word in their document. There is very little that can go wrong in this program, and most of it comes down to design decisions and programming on my part, not user error.