CS 32, Winter 2021
Project 4: Wurd

Due: 11 PM, Thursday, March 11

Make sure to read the entire document (especially Requirements and Other Thoughts) before starting your project.

Introduction

Before writing a single line of code or reading the rest of this document, you MUST first read AND THEN RE-READ the Requirements and Other Thoughts section. Print out this page, tape it to your wall or on the mattress above your bunk bed, etc. And read it over and over.

The NachenSmall Software Corporation, which has traditionally only built software for running senior-citizen bingo games, has decided to pivot into a new area. They've decided to disrupt the word processor market and build a retro, console-based text editor called Wurd.

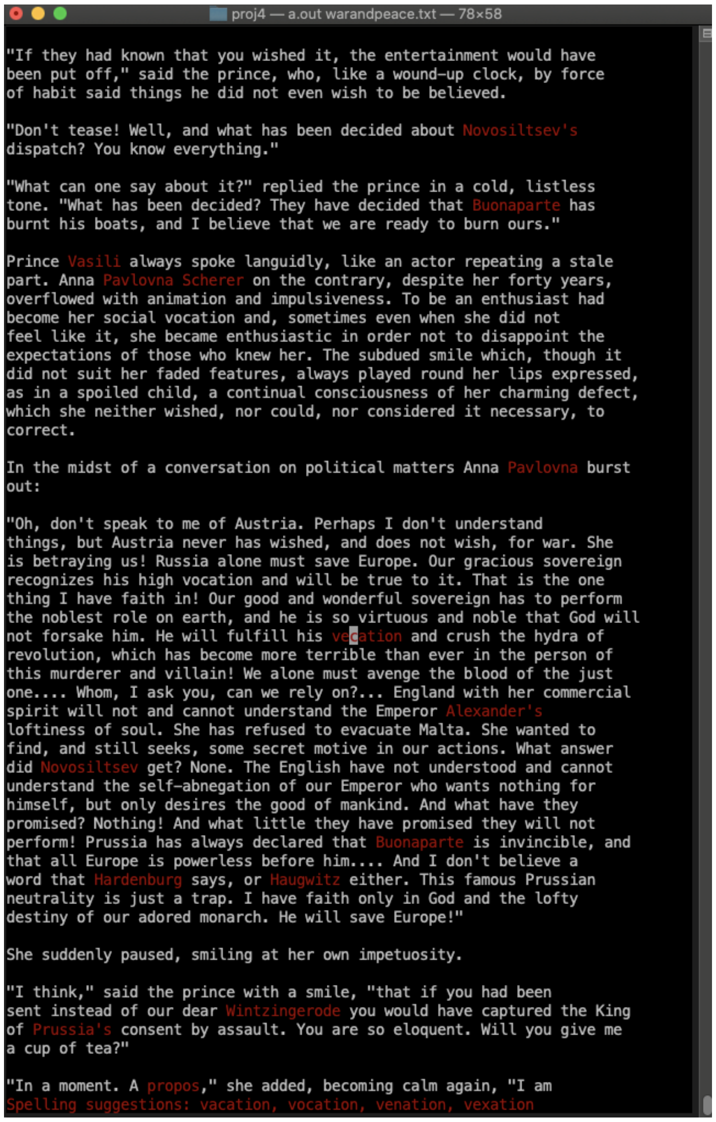
The evil co-CEOs, Carey and David (not to be confused with Harry and David, the purveyors of yummy gourmet gift baskets) have become increasingly frustrated with today's complex, slow word processors and are looking for an alternative that's simple and fast. They want three basic features:

- The ability to edit text documents of virtually any size
- Basic undo capabilities
- Real-time spell-checking with spelling suggestions

Given their eccentric ways, Carey and David initially asked a student at UC Berkeley to build their new word processor. The student was able to complete the user-interface (textual GUI) portion of the word processor, but then he resigned to go on a six-month ganja retreat in the Santa Cruz mountains (you know those UCB undergrads).

Unfortunately, he was therefore unable to complete the core text editor, undo, or spell-checking classes. So Carey and David will be providing you with his simple text-based GUI for you to build off of. Your job in this year's Project 4 is to build the three classes required to complete the new wurd-processor: the core editor class, the undo class, and the spell-checker class.

Here's a mocked up screen-shot of Wurd, shown editing the classic novel *War and Peace* by Tolstoy:



```
"If they had known that you wished it, the entertainment would have
been put off," said the prince, who, like a wound-up clock, by force
of habit said things he did not even wish to be believed.

"Don't tease! Well, and what has been decided about Novosiltsev's
dispatch? You know everything."

"What can one say about it?" replied the prince in a cold, listless
tone. "What has been decided? They have decided that Buonaparte has
burnt his boats, and I believe that we are ready to burn ours."

Prince Vasili always spoke languidly, like an actor repeating a stale
part. Anna Pavlovna Scherer on the contrary, despite her forty years,
overflowed with animation and impulsiveness. To be an enthusiast had
become her social vocation and, sometimes even when she did not
feel like it, she became enthusiastic in order not to disappoint the
expectations of those who knew her. The subdued smile which, though it
did not suit her faded features, always played round her lips expressed,
as in a spoiled child, a continual consciousness of her charming defect,
which she neither wished, nor could, nor considered it necessary, to
correct.

In the midst of a conversation on political matters Anna Pavlovna burst
out:

"Oh, don't speak to me of Austria. Perhaps I don't understand
things, but Austria never has wished, and does not wish, for war. She
is betraying us! Russia alone must save Europe. Our gracious sovereign
recognizes his high vocation and will be true to it. That is the one
thing I have faith in! Our good and wonderful sovereign has to perform
the noblest role on earth, and he is so virtuous and noble that God will
not forsake him. He will fulfill his vecation and crush the hydra of
revolution, which has become more terrible than ever in the person of
this murderer and villain! We alone must avenge the blood of the just
one.... Whom, I ask you, can we rely on?... England with her commercial
spirit will not and cannot understand the Emperor Alexander's
loftiness of soul. She has refused to evacuate Malta. She wanted to
find, and still seeks, some secret motive in our actions. What answer
did Novosiltsev get? None. The English have not understood and cannot
understand the self-abnegation of our Emperor who wants nothing for
himself, but only desires the good of mankind. And what have they
promised? Nothing! And what little they have promised they will not
perform! Prussia has always declared that Buonaparte is invincible, and
that all Europe is powerless before him.... And I don't believe a
word that Hardenburg says, or Haugwitz either. This famous Prussian
neutrality is just a trap. I have faith only in God and the lofty
destiny of our adored monarch. He will save Europe!"

She suddenly paused, smiling at her own impetuosity.

"I think," said the prince with a smile, "that if you had been
sent instead of our dear Wintzingerode you would have captured the King
of Prussia's consent by assault. You are so eloquent. Will you give me
a cup of tea?"

"In a moment. A propos," she added, becoming calm again, "I am
Spelling suggestions: vacation, vocation, venation, vexation
```

There are a couple of things to notice:

1. All words that are misspelled are highlighted in red. Carey and David are horrible spellerz and therefore want real-time highlighting of misspelled words. They've agreed to provide you with a text file containing over 100k words for this spell-checker.

2. They want Wurd to provide real-time spelling suggestions as you move the cursor over a word. Notice that the cursor is hovering over the misspelled word “vecation” about midway down the screen. At the bottom of the screen, they want Wurd to display spelling suggestions - in this case it’s suggesting: vacation, vocation, venation, and vexation. (Note: These are all words that are exactly one character different from the misspelled word vecation.)

What Do You Need to Do?

Question: So, at a high level, what do you need to build to complete Project 4?

Answer: You’ll be building three complete classes, detailed below:

Class #1: You need to build a class named ***StudentTextEditor*** that implements the core editing functionality of a text editor (allowing you to insert and delete characters, load and save text files, and undo your changes). **This class *MUST* be derived from our TextEditor base class or you will get a zero on this project.**

Class #2: You need to build a class named ***StudentUndo*** that implements undo capabilities for your StudentTextEditor class. **This class *MUST* be derived from our Undo base class or you will get a zero on this project.**

Class #3: You need to build a class named ***StudentSpellCheck*** that implements spell check and spelling suggestion functionality based on our provided dictionary. **This class *MUST* be derived from our SpellCheck base class or you will get a zero on this project.**

You will find all of the gory details below in the section called Details.

What Will We Provide

We will provide the following for your use:

- A main.cpp file that has a main() function that runs the word processor. **YOU MUST NOT MODIFY THIS FILE (except in a limited way detailed below) AS YOU WILL NOT TURN IT IN WITH YOUR SOLUTION.**
- An EditorGui.h file which contains the graphical user interface that displays the text editor on the screen, and other files starting with lower case letters that support the graphics. **YOU MUST NOT MODIFY THESE FILES AS YOU WILL NOT TURN IT IN WITH YOUR SOLUTION.**

- A TextEditor.h file which contains the TextEditor base class that you must derive your StudentTextEditor from. **YOU MUST NOT MODIFY THIS FILE AS YOU WILL NOT TURN IT IN WITH YOUR SOLUTION.**
- An Undo.h file which contains the Undo base class that you must derive your StudentUndo from. **YOU MUST NOT MODIFY THIS FILE AS YOU WILL NOT TURN IT IN WITH YOUR SOLUTION.**
- A SpellCheck.h file which contains the SpellCheck base class that you must derive your StudentSpellCheck from. **YOU MUST NOT MODIFY THIS FILE AS YOU WILL NOT TURN IT IN WITH YOUR SOLUTION.**
- A dictionary.txt file which contains a list of about 110,000 English words (without definitions - just the properly spelled words)

In main.cpp, you may modify the initialization of DICTIONARYPATH to specify the location of a dictionary file; to minimize problems some people have, you should specify a full path name like "C:/CS32/p4/dictionary.txt" (note the forward slashes) or "/Users/alex/cs32/proj4/dictionary.txt". You may also modify the colors initializing FOREGROUND_COLOR, BACKGROUND_COLOR, and HIGHLIGHT_COLOR if you don't like Carey's choice of colors.

If you define your classes correctly and derive them from our base classes, they should work perfectly with our GUI class and you'll create your own awesome Wurd processor!

Even more cool, once you get your Wurd processor sort of working, you can use it to edit and save your Wurd processor's source code. That's called bootstrapping! Weird huh, editing your own Wurd processor's source code with your own Wurd processor built from it?! MIND BLOWN!

But be careful! If your Wurd processor has bugs and you use it to edit your own source code, you could corrupt your source! EEK! So if you try this, back up your source code first!

Details

StudentTextEditor Class

You must build a class called StudentTextEditor which implements the biggest part of the text editor. This class doesn't directly interact with the user. Instead it provides a set of methods to insert characters, delete characters, move the editing position (where new characters will be inserted, or existing characters will be deleted) within the editor, etc. Our provided EditorGui class is responsible for getting input from the user and then calling your StudentTextEditor methods to store the text the user is typing, delete text, etc. Here's a simplified example of how your StudentTextEditor class (which is derived from the provided TextEditor class) could be used by our GUI class:

```
class EditorGui {
public:
        EditorGui() {
                // construct a new student text editor object.
                text_editor = createTextEditor(...);
        }

        void getKeyFromUserAndProcessIt() {
                char ch;
                cin.get(ch);  // this is simplified; it's not actually read this way
                if (isAlphaNumeric(ch)) text_editor->insert(ch);
                else if (isBackspace(ch)) text_editor->backspace();
                else if (isDelete(ch)) text_editor->delete();
                else if (isUpArrowKey(ch)) text_editor->move(UP);
                ...
                else if (isUndoKey(ch)) text_editor->undo();
                …
        }

private:
        TextEditor *text_editor;  // Base class pointer to our TextEditor base
};
```

Your StudentTextEditor class:

- **MUST** be derived from our TextEditor base class.
- **MUST NOT** contain ANY member variables of type std::vector or std::deque or std::array
- **MUST NOT** add any new public member functions or variables
- **MUST** meet the big-O requirements detailed below for each of its methods
- MAY use the STL list, map, and set types
- MAY have any private methods/variables it needs (you can add these) that are not forbidden above

So what are the essential data components of a text editor? Here are the essentials:

- The lines of text being edited (in some data structure)
- The current editing position in the text, i.e., the row and column where new characters will be inserted or existing characters will be deleted/backspaced
- Data to facilitate undo operations; you will use your StudentUndo class for this purpose

You may find it useful to also store other data items in your class (which you will probably need to do to meet the requirements of this spec), but the items above are the minimal required for a basic text editor.

Your StudentTextEditor class must have the following methods:

StudentTextEditor(Undo* undo)

You must define a constructor that accepts a pointer to an Undo object (or something derived from Undo). The constructor must pass this undo pointer to the TextEditor base class's constructor, where the undo object's pointer will be stored. The constructor must initialize the current editing position (the position where the cursor would be displayed in the file) to row 0, column 0; that is, the first character in the edited file. It may also do any other initialization that it needs to do to manage your text editor's data structures. **This method must run in O(1) time.**

~StudentTextEditor()

You must define a destructor for your text editor class that frees all memory used by your object. It MUST NOT free the undo object via the pointer that was passed in, as this will be performed by our provided GUI class. **This method must run in O(N) time where N is the number of lines in the file currently being edited.**

bool load(std::string file)

You must implement a load method to load the contents of a text file off disk into the editor. The file string will contain a full path and filename, like "C:/cs32/proj4/war-and-peace.txt" (notice the use for forward slashes, even on a Windows system). Your method must open the specified file using C++ file I/O classes[1] and read its contents into its data structure(s). As you read lines from the file, each line might or might not have a carriage return character ('\r') just before the newline at the end of the line. You must strip out such a carriage return before adding the line to your internal data structures. If the specified file cannot be found, then your load method should do nothing and return false. If you were already editing an existing file and the specified file can be loaded, then the old contents of the text editor must be reset (described below) and upon completion of the load method, your text editor must contain the contents of the new file. After successfully loading the specified file, your load method must reset the current editing position to the beginning row/column of the file and return true. **This method must run in O(M+N+U) time where M is the number of characters in the editor currently being edited, N is the number of characters in the new file being loaded, and U is the number of items in the undo stack.**

bool save(std::string file)

You must implement a save method to save the contents of the text editor to a file on your disk. The file string will contain a full path and filename, like "C:/cs32/proj4/war-and-peace.txt" where you want to save the lines you're currently editing. Your method must open the specified file using C++ file IO classes[2] and write the lines in the text editor to the file, overwriting any previous data in the file with the new contents. After each line written to the file, you must

[1] See the File I/O writeup on the main CS 32 web page for details on how to read/write text files.
[2] See the File I/O writeup on the main CS 32 web page for details on how to read/write text files.

append a newline ('\n') character. If the specified output file cannot be opened, then your save method must return false. Otherwise, after successfully saving the lines in the editor to the specified file, your method must return true. **This method must run in O(M) time where M is the number of characters in the editor currently being edited.**

void reset()

The reset method clears the text editor's contents and resets the editing position to the top of the file (row 0, column 0). After the reset method is called, there should be no text within the editor. The reset method must also clear the undo state, so no undos are possible after the reset. **This operation must run in O(N+U) time, where N is the number of rows being edited and U is the number of items in the undo stack.**

void insert(char ch)

The insert command inserts a new character at the current editing position in the editor, moving all characters to the right of the editing position right by one. After the insertion, the current editing position must be moved right by one spot. In addition to performing the insert operation, this method must also tell its Undo member variable to track the insertion so it may be undone later by the user (if requested). So, if the edited text is:

> "I like traffic lights."
>
> with the current editing position at the underlined character (t in traffic), then calling the insert method with a character of 's' must change the line to:
>
> "I like straffic lights."
>
> with the editing position moved right to the t in traffic.

If the user inserts a tab character, your editor must replace the tab and instead insert exactly 4 spaces into the current line at the current editing position, and then move the current editing position right by 4 spots.

Insertion of a character into a line must run in O(L) time where L is the length of the line of text containing the current editing position.

void enter()

The enter command inserts a line break at the current editing position in the editor, splitting the current line into two parts. After the line break, the editing position must be moved one line down, to the first column of the newly split line. In addition to performing the line break operation, this method must also tell its Undo member variable to track the line break so it may be undone later by the user (if requested). So, if the edited text is:

I like traffic lights.
But only when they’re green.

with the current editing position at the underlined character (r in traffic), then calling the enter method must change the line to:

I like t
raffic lights.
But only when they’re green.

with the editing position at the front of the newly split second line.

If the current editing position is just past the end of the last line of the text, a new empty line is added to the end of the text, and the editing position is at the first column of that empty line.

Using the enter command anywhere on a line must run in O(L) time where L is the length of the line of text containing the current editing position. This command must not have a runtime that depends on the number of lines being edited!

void del()

The del command is used to delete the character at the current editing position in the editor. In addition to performing the delete operation, it must also tell its Undo member variable to track the deletion so it may be undone later by the user (if requested). The del command must have the following behaviors:

- If the editing position is on a valid character, the del command must delete the current character from the current line without changing the editing position. e.g., if the edited text is:

  I like traffic lights.

  with the current editing position at the underlined character (t in traffic), then calling the del method must change the line to:

  I like raffic lights.

  with the editing position at the r in raffic.

- If the editing position is just after the last character on a line, the del command must merge the current line with the line below (if one exists), leaving the editing position unchanged. So, if these were your three lines:

I like_

traffic lights
but only when they’re green

with the underline shows the current editing position (after the end of the top line), then after calling the del method, the resulting lines must look like this:

I liketraffic lights
but only when they’re green

with the editing position still just after the e (now at the t in traffic), unchanged from where it was originally. Note that there’s no space between the merged lines. Since the second line will be merged with the current line, the original second line (“traffic lights”) must be deleted from your editor after it’s been merged with the current line.

- If the editing position is just after the last character on the last line, then the del command does nothing (since there’s nothing to delete)

Deletion of a character in the middle of a line must run in O(L) time where L is the length of the line of text containing the current editing position. Deletion at the end of a line resulting a merge must run in O(L1+L2) where L1 is the length of the current line of text at the editing position and L2 is the length of the next line in the editor. This command must not have a runtime that depends on the number of lines being edited!

void backspace()

The backspace command is used to delete a character just before the current editing position in the editor. In addition to performing the backspace operation, it must also tell its Undo member variable to track the backspace so it may be undone later by the user (if requested). The backspace command must have the following behaviors:

- If the current editing position’s column is > 0 (meaning that the editing position is past the first character of the nonempty current line), the backspace command must delete the character just before the editing position from the current line, then move the editing position left by one. e.g., if the edited text is:

I like traffic lights.

with the current editing position at the underlined character (e in like), then calling the backspace method must change the line to:

I lie traffic lights.

with the editing position changed to be one spot to the left, but still at the e in like.

- If the editing position is at the first character of a line or the line is empty, the backspace command must merge the current line with the line above it (if one exists), and change the editing position to be at the merge point of the two lines (which, if the editing position was on an empty line, would be just past the end of the merged line). So, if these were your three lines:

  I like
  traffic lights
  but only when they're green

  and the underscore shows the current editing position (at the front of the second line), then after calling the backspace method, the resulting lines must look like this:

  I liketraffic lights
  but only when they're green

  with the editing position just after the e (on the t in traffic), moved to the line above that was merged into. Note that there's no space between the merged lines. Since the current line will be concatenated to the end of the previous line, the original second line ("traffic lights") must be deleted from your editor after it's been merged with the previous line.

- If the editing position is on the first character of the first line, then the backspace command does nothing (since there's nothing to backspace over)

Backspacing that does not result in a merge must run in O(L) time where L is the length of the line of text containing the current editing position. Backspacing at the front of a line resulting in a merge with the previous line must run in O(L1+L2) where L1 is the length of the line of text containing the current editing position and L2 is the length of the previous line in the editor. This command must not have a runtime that depends on the number of lines being edited!

void move(Dir dir)

Your StudentTextEditor class must maintain a current editing position (where the user is currently editing/deleting text). The move method is used to adjust the editing position, for instance, when the user presses the up/down/left/right keys to move the cursor on the screen. The argument is of a public enumerated type Dir declared in TextEditor with constants Dir::UP, Dir::DOWN, etc. Valid directions include:

- Dir::UP - moves the editing position up one line; if the editing position is already on the top line, then this does nothing
- Dir::DOWN - moves the editing position down one line; if the editing position is already on the bottom line, then this does nothing

- Dir::LEFT - moves the editing position left by one character; if the editing position is already on the first character of the line, then this command positions the cursor *after* the last character on the previous line. If the editing position is already on the first character of the top line, then this command does nothing
- Dir::RIGHT - moves the editing position right by one character; if the editing position is just after the last character of the current line, then this command positions the cursor on the first column of the next line. If the editing position is already after the last character of the last line, then this command does nothing
- Dir::HOME - moves the editing position to the first character on the current line
- Dir::END - moves the editing position *just after* the last character on the current line

Each of these operations must run in O(1) time, where the operation's runtime does not depend on the number of lines being edited or the length of the current line.

void getPos(int& row, int& col) const

The getPos method sets its parameters to the row and column of the editor's current editing position.

This operation must run in O(1) time, where the operation's runtime does not depend on the number of lines being edited or the length of the current line.

int getLines(int startRow, int numRows, std::vector<std::string>& lines) const

If startRow or numRows is negative, or if startRow is greater than the number of lines in the text, return -1 without changing the lines parameter. Otherwise, the getLines method retrieves the specified rows of text from the text editor and places them into the lines variable, clearing any previous data that was in the lines variable before adding the requested lines. If fewer than numRows rows are available starting at row startRow, then only the available lines must be added to the lines parameter; if startRow is equal to the number of lines in the text, the lines parameter will be empty upon return. This function returns the number of lines in the lines parameter upon return.

So calling getLines(2, 10, lines) on the following text in the editor:

```
this
is
a
test
of the emergency broadcasting system.
```

Would place the following into the lines vector:

```
a
```

test
of the emergency broadcast system.

Let oldR be lines.size() at the time this function was entered. This operation must run in O(oldR + abs(current row number - startRow) + numRows*L) time, where L is the average line length. The runtime must not depend on the number of lines being edited.

void undo()

The undo method must undo the last operation the user performed and reposition the cursor as described below. The TextEditor base class has its own Undo member variable, and your StudentTextEditor's undo method must ask this undo member variable (obtained by calling TextEdit's protected getUndo() method) for the most recent undo operation and then apply it (assuming there is something to undo).

Each undo operation will contain:

- An undo action (e.g., insert characters back into the document, delete characters, etc.)
- A position (row, column) where to insert/delete the specified characters on behalf of the undo command
- One of the following two items:
    - One more characters to insert back into the editor (e.g., the characters that were just deleted)
    - A count of how many characters to delete from editor (e.g., if one or more characters were just inserted and the undo must delete them)

Here are the possible undo operations your StudentTextEditor must handle. Undo::Action is an enumerated type with constants INSERT, DELETE, etc.:

- If the undo action is Undo::Action::INSERT then your undo method must position the cursor on the specified row, column and then insert all of the characters in the specified undo string back into the document.
- If the undo action is Undo::Action::DELETE then your undo method must position the cursor on the specified row, column and then delete the specified count of characters.
- If the undo action is Undo::Action::SPLIT then your undo method must position the cursor on the specified row, column and then add a line break.
- If the undo action is Undo::Action::JOIN then your undo method must position the cursor on the specified row, column (the row, column will always be at the end of a line, which you will join with the line below) and then join two lines together (as if the user pressed the delete key at the end of the line).
- If the undo action is Undo::Action::ERROR then your undo method must do nothing, as the undo stack is empty (there are no further changes to undo).

In all cases, after completion of the undo operation as described above, the cursor must be set to the specified row, column returned by the undo get command.

For more information about how the undo command works, please see the Undo section. NOTE: There is no undo possible for the changes restored by the undo command (i.e., no redo, no undoing the undo).

This method must run in a time proportional to the nature of the undo operation plus the distance from the current cursor position to the position of the operation. For example, if the undo is of a deleted character, then its runtime must be the cost to re-insert the deleted character into the editor. If the undo is of a line break, then the runtime must be the cost to re-merge the line that was broken into two parts, etc.

StudentUndo Class

The StudentUndo class is responsible for managing the data structures required for undo operations for your word processor. At its simplest, an undo class can be implemented with a simple *stack*. The item at the top of the stack contains the last operation the editor might wish to undo (say a deletion or insertion of a character), and the item at the bottom of the stack contains the oldest change that was made (and therefore the last thing to be undone).

Your StudentUndo class:

- **MUST** be derived from our Undo base class.
- **MUST** meet the big-O requirements detailed below for each of its methods
- **MUST NOT** add any new public member functions or variables
- MAY use any STL classes
- MAY have private methods/variables it needs (you can add these)

Your StudentUndo class has three primary methods:

- **submit**: Allows your StudentTextEditor to submit a user's change to the undo system so it can later be undone
- **get**: Get from the undo system the most recent change that was submitted to the undo system (so it can be undone in the StudentTextEditor)
- **clear**: Remove all submissions from the undo system so it is empty

It may also have a constructor and destructor, as necessary.

void submit(Undo::Action action, int row, int col, char ch)

Your text editor needs to submit several pieces of information to the undo object every time the user makes a change to the currently edited document via the StudentTextEditor class:

- The action the user just did in the text editor (e.g., the user just inserted a character into the document, the user just deleted a character from the document, etc.). These include:
    - **INSERT**: The user just inserted a character into the document. (If the user later chooses to undo that operation, the undo system will need to provide the details needed to delete that character)
    - **DELETE**: The user just deleted a character from the document. (If the user later chooses to undo that operation, the undo system will need to provide the details to insert that specific character back)
    - **JOIN**: The user just joined two lines together in the document, by hitting the delete key at the end of a line, or the backspace key at the front of a line. (If the user later chooses to undo that operation, the undo system will need to provide the details of where to split that joined line back into two lines)
    - **SPLIT**: The user just hit the enter key, splitting a line by adding a line break into the document (if the user later chooses to undo that operation, the undo system will need to provide the details of where to remove the line break from the document, rejoining the two lines where the break was added)
- The row and column in the edited document where the user's action happened (this is needed to know where to undo the change)
- The character involved in the user's action. For example, if the user just deleted an 'a' from the document, then the undo system must track that the letter 'a' was deleted, so if the user asks it to undo this operation, the undo system knows that an 'a' needs to be inserted back into the document at the proper location.

Here's how your StudentEditor class must use the Undo stack's submit method:

- Any time the user **inserts a character into a document**, your text editor must call the Undo class's submit method with the following information:
    - An action of Undo::Action::INSERT
    - The row and column in the document for the editing position AFTER the new character was inserted by your text editor to the document (on the column after the inserted character)
    - The character that was inserted, e.g., a space or the letter 'a'
- Any time the user **uses the delete key to delete a character from other than just past the end of a line**, your text editor must call the Undo class's submit method with the following information:
    - An action of Undo::Action::DELETE
    - The row and column in the document for the editing position AFTER the delete key was processed by your text editor
    - The character that was deleted, e.g., a space or the letter 'a'
- Any time the user **uses the backspace key to delete a character from other than the beginning of a line**, your text editor must call the undo class's submit method with the following information:
    - An action of Undo::Action::DELETE

    - The row and column in the document for the editing position AFTER the backspace key was processed by your text editor (one position left of the editing position before the backspace was pressed)
    - The character that was backspaced over, e.g., a space or the letter ‘a’
- Any time the user **uses the delete key at the end of a line**, resulting in a join of the current line with the line below it, your text editor must call the undo class’s submit method with the following information:
    - An action of Undo::Action::JOIN
    - The row and column in the document for the editing position AFTER the delete key was processed by your text editor (the cursor will on the line where the delete key was pressed)
- Any time the user **uses the backspace key at the front of a line**, resulting in a join of the current line with the previous line, your text editor must call the undo class’s submit method with the following information:
    - An action of Undo::Action::JOIN
    - The row and column in the document for the editing position AFTER the backspace key was processed by your text editor (the cursor will on the line above where the backspace was pressed)
- Any time the user **presses the enter key**, resulting in line break being added to the document at the current editing position, your text editor must call the undo class’s submit method with the following information:
    - An action of Undo::Action::SPLIT
    - The row and column in the document for the editing position BEFORE the enter key was processed by your text editor (the cursor will on the line where the enter key was pressed, NOT on the following line where the cursor will move *after* the enter key was processed)

WARNING: Note that for the enter key case, the row and column passed in must be the editing position BEFORE the enter key was processed by your text editor, NOT afterward.

There are **three** additional rules that you must follow when implementing your undo submit method:

- For undoing **delete** operations that consecutively occur at the same spot in the edited file, you must “batch” them together so they are treated as a single undo operation. For example, given this text, with the user’s cursor at the point of the underline:

    Go bruins!

  If the user hits the delete key once, they’d see:

    Go ruins!

  If the user hits delete two more times, they’d see:

Go ins!

Later if the user were to undo with a single ctrl-z command, the undo operation must restore all three consecutive deleted characters at once to:

Go bruins!

With the cursor on the original spot (‘b’) where the consecutive deletions started.

In contrast, imagine the user does two consecutive deletions that are not at the same place, the first at the b and the second at the i:

Go runs!

In this case, the first undo operation will restore the i:

Go ruins!

And the second will restore the b:

Go bruins!

This rule does NOT hold if you press delete at the end of a line (to join the line below with the current line). A deletion at the end of a line is always considered a separate undo operation and must never be merged with another undo operation.

- For undoing a series of **backspace** operations that backspace over a consecutive series of characters on the same line, you must “batch” them together so they are treated as a single undo operation. For example, given this text, with the user’s cursor at the point of the underline:

  Go bruins!

  Hitting backspace five times in a row will result in this text:

  Gons!

  Hitting ctrl-z to undo this change should instantly restore the text to:

  Go bruins!

  With the cursor placed at the front of the set of consecutive backspace operations (e.g., on the space before “bruins!”).

This rule does NOT hold if you press backspace at the front of a line (to join the current line with the one above). A backspace at the front of a line (a join) is always considered a separate undo operation and must never be merged with another undo operation.

- For undoing **insert** operations that consecutively occur in adjacent spots (at column p, and then at column p+1, p+2, etc.) in the edited file, you must "batch" them together so they are treated as a single undo operation. For example, given this text, with the user's cursor at the point of the underline:

  Go bruins!

  If the user were to type the letters 'b' then 'i' then 'g' followed by a space, the resulting text would be:

  Go big bruins!

  Later if the user were to undo with a single ctrl-z command, the undo operation must remove all four inserted characters at once to:

  Go bruins!

  Placing the cursor at the position where the first of the consecutive changes was made.

  This is because all four inserted characters came one after the other on the same line. Had they been inserted in non-adjacent locations, or had some other operation come in between them (e.g., a line break, a deletion, an insertion in some other location that is not adjacent), then these must result in separate undo operations.

If you have trouble figuring out how to do "batching" then try to at least be able to do single-character undos.

This method must run in O(1) time in the average case, understanding that it may increase to O(current line length where the operation occurred) infrequently.

Undo::Action get(int& row, int& col, int& count, std::string& text)

When the user hits Ctrl-Z in our editor GUI, we will call the undo method in your StudentTextEditor, causing an undo operation to take place. At such a time, your text editor's undo method must call its undo object's[3] get method in order to obtain the most recently submitted undo operation. The get method returns the following information in its parameters:

- An action to take to undo a previous change:
    - If the original change made to the document was an insertion of characters, the action returned by the get method must be Undo::Action::**DELETE**.
    - If the original change made to the document was a deletion of characters using the delete key or backspace, the action returned by the get method must be Undo::Action::**INSERT**.
    - If the original change made to the document was to introduce a line break, then the action returned by the get method must be Undo::Action::**JOIN**.
    - If the original change made to the document was to introduce join two lines (with backspace or delete), then the action returned by the get method must be Undo::Action::**SPLIT**.
- The row and column in the edited document where the restoration must happen:
    - For an undo action of Undo::Action::**INSERT** (to undo an original delete/backspace), the row, col must be the *starting* location where the deleted text should re-inserted.
    - For an undo action of Undo::Action::**DELETE** (to undo an original insertion of one or more characters), the row, col must be the *starting* location where the text that was inserted should be removed.
    - For an undo action of Undo::Action::**JOIN** (to join two lines that were broken via an enter key), the row, col must be the location where the enter key was hit (before the line was split into two).
    - For an undo action of Undo::Action::**SPLIT** (to break one line back into two that were joined via a deletion at the end of a line, or a backspace at the front of a line), the row, col must be:
        - If the Undo::Action::**SPLIT** is to undo a delete at the end of a line (joining a line with the line below it), then the row, col must be the position at the end of the top line BEFORE the delete key joined the two lines
        - If the Undo::Action::**SPLIT** is to undo a backspace at the front of a line (joining a line with the line above it), then the row, col must be the position at the end of the line above AFTER the backspace was processed and the lines were joined

      The row, col will be such that in both cases (whether the user had originally hit a delete or a backspace), adding a line break at the row, col would re-split the line as if the undo hadn't occurred.
- A count of how many characters were involved:
    - For Undo::Action::**DELETE** actions, the count must be the number of characters to delete
    - For all other actions (**INSERT**, **JOIN**, **SPLIT**), the count must be 1
- A text string:

[3] A pointer to the undo object is held in the TextEditor base class. It can be retrieved by your StudentTextEditor class via TextEditor's getUndo() method.

    - For Undo::Action::**INSERT** actions, a string of zero or more characters to insert back into the document (e.g., when the user deleted/backspaced over one or more characters) at the specified row, col. For example, if the user just deleted the word “bruins!” from the document, then the string will be “bruins!”
    - For all other actions (**DELETE**, **JOIN**, **SPLIT**), the string must be empty

The undo operation is removed from the undo stack. If the undo stack was empty when this method was called, it must return an action of Undo::Action::**ERROR**, and the other parameters (e.g., row, col) must remain unchanged.

This method must run in O(1) time in the average case, understanding that it may increase to O(length of deleted characters that need to be restored) for restoration cases.

void clear()

This method must clear out your entire undo stack, making it empty.

It must run in O(N) time where N is the number of elements in the undo stack.

StudentSpellCheck Class

The StudentSpellCheck class is responsible for providing spell-checking capabilities and also providing suggestions for misspelled words.

Your StudentSpellCheck class:

- **MUST** be derived from our SpellCheck base class.
- **MUST** meet the big-O requirements detailed below for each of its methods
- **MUST** treat an apostrophe (’) as a letter that must be spell-checked
- **MUST** be case insensitive for all word lookups
- **MUST NOT** use any STL containers EXCEPT for STL vectors and strings
- **MUST NOT** add any new public member functions or variables
- MAY have private methods/variables it needs (you can add these)

Your spell check class has three primary methods:

- **load:** Loads words from the “dictionary” data file which contains an ordered list of english words.
- **spellCheck**: Given a word, determines if it’s in the dictionary, and if not, returns zero or more spelling suggestions
- **spellCheckLine**: Given a line of text, determines the starting/ending positions of all misspelled words in the line

It may also have a constructor and destructor, as necessary.

Your spell check class must use a *trie* data structure (we'll tell you more about a trie in a bit) that you code yourself to hold its words, or you will not receive credit for this part of the project.

bool load(std::string dictionaryFile)

This command must load the specified dictionary file specified in the dictionaryFile parameter into a trie data structure. The trie must be a private member variable of your class. For more details on what a trie data structure is (guess what, it's a type of tree), please see this explanation.

Each line of the file represents one dictionary entry. If stripping all non-letter, non-apostrophe characters from the line leaves at least one character, those remaining characters on the line form a word to add to the trie if it's not already present.

Pro tip: When building the dictionary, if you either upper or lower-case the letters (whatever you do, be consistent) it makes searches simpler.

Your method must return true if the dictionary was loaded properly, and false if it could not be loaded/opened.

This function must run in O(N) time where N is the number of lines in the dictionary file, assuming that there's a constant upper bound on the length of an input line.

bool spellCheck(std::string word, int maxSuggestions, std::vector<std::string>& suggestions)

Given a word, the spellCheck function must check for it in your trie in a case-insensitive manner (so "HeLlO" should return true if "hello" is in the dictionary) and return true if the word is found, and false otherwise. If the specified word cannot be found, then your function must clear the suggestions parameter, and then add zero or more spelling suggestions, up to a count of maxSuggestions, into the suggestions vector parameter.

A word from the dictionary is a candidate for a spelling suggestion if it is the same length as the passed-in word, and differs by at most one character. So, for example, if you passed in a word "donet" the proper suggestions from our dictionary would be "don't" and "donut". For the purposes of this assignment, an improper suggestion would be "done". Or if the word "xole" was passed in, the proper suggestions would be "bole", "cole", "dole", "hole", etc.

Your method must return true if the word is in the dictionary, and false otherwise.

Let oldS be suggestions.size() at the time this function was entered. This function must run in O(oldS + L^2 + maxSuggestions) time where L is the length of the word being searched for.

void spellCheckLine(const std::string& line, std::vector<Position>& problems)

Position is defined as follows:

```
struct Position {
  int start;
  int end; // inclusive
};
```

This function is responsible for spell-checking a full line of text and placing the starting/ending position of each misspelled word (i.e., word not in the dictionary) into the problems parameter vector. If no misspelled words are found, the problems vector must be empty upon return. To spell check the line, the method or one of its helpers must break the line up into distinct words. A word is comprised of one or more letters or the apostrophe character (as in don't) that are contiguous. All other characters, like dashes, periods, commas, double quotes, etc., should be considered separators like a space and must not be considered to be part of a word. So given this line of text, with letter positions shown vertically below it:

```
Diz"iz-a tezt.
00000000001111
01234567890123
```

The method would first break up the line into the following words:

- Diz: positions 0 → 2
- iz: positions 4 → 5
- a: position 7 → 7
- tezt: position 9 → 12

and look up each word in the dictionary in a case-insensitive manner. In this example, the words Diz, iz and tezt, are all misspelled and would not be found in the dictionary's trie. Thus the output of this function would be the following vector:

{0, 2}
{4, 5}
{9, 12}

which indicates the starting and ending positions (inclusive) of the misspelled words.

Let oldP be problems.size() at the time this function was entered. This function must run in O(oldP+S+W*L) time where S is the length of the line passed in, W is the number of words in the line, and L is the maximum length of a checked word.

Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

- Back up your code to Google Drive, iCloud or some other cloud service every time you make progress. WE WILL NOT ACCEPT CRASHED COMPUTERS/LOST FILES AS AN EXCUSE.
- If you use the Visual Studio or Xcode debugger, you will probably shave about 50% of your development time off this project. So use the debugger.
- In Visual C++, make sure to change your project from UNICODE to Multi Byte Character set, by going to Project / Properties / Configuration Properties / General / Character Set
- The entire project can be completed in under 600 lines of C++ code beyond what we've already written for you, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.
- Let XXX and YYY be distinct names in { TextEditor, Undo, SpellCheck }. Your StudentXXX.h and StudentXXX.cpp **must not** refer to your StudentYYY.h or StudentYYY.cpp; they may refer to our XXX.h and YYY.h. For example, your StudentTextEditor.cpp **must not** include your StudentUndo.h, although it may include our Undo.h and your StudentTextEditor.h.
- You must not modify any of the code in the files we provide you, as you will not turn them in — if you modify them, we will not see those changes. We will incorporate the required files that you submit into a project with special test versions of the other files.
- You must derive your classes from our base classes or you will get a zero on the project.
- You must not add any public member variables/functions to your derived classes. You may add private member variables/helper methods.
- Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. How will you use these data structures? Plan before you program!
- Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
- Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
- To get full credit, you may use only those STL containers (e.g., vector, list) that are explicitly permitted for each class. However, if you're having trouble building a data structure from scratch, feel free to use the STL containers to help you make progress.

Using banned STL containers will result in a point deduction, potentially taking your score to zero on the violating class.

If you don't think you'll be able to finish this project, then take some shortcuts. For example, if you can't get your StudentSpellCheck class working with a hand-built Trie, use the STL set class temporarily so that you can proceed with implementing other classes, and go back to fixing your StudentSpellCheck class later.

You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to complete a class (e.g., StudentUndo), we will provide a correct version of that class and test it with the rest of your program (e.g., we'll test our correct StudentUndo with your StudentTextEditor). If you implemented the rest of the program properly (and derived your classes from our base classes) our version of the StudentUndo class should work perfectly with your version of the StudentTextEditor class and we can give you credit for those parts of the project you completed.

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors under both g32 and either Visual Studio or clang++!

What to Turn In

You must turn in six files.

StudentTextEditor.h	Contains your StudentTextEditor class declaration
StudentTextEditor.cpp	Contains your StudentTextEditor class implementation
StudentUndo.h	Contains your StudentUndo class declaration
StudentUndo.cpp	Contains your StudentUndo class implementation
StudentSpellCheck.h	Contains your StudentSpellCheck class declaration
StudentSpellCheck.cpp	Contains your StudentSpellCheck class implementation

You are to define your class declarations and all member function implementations directly within the specified .h and .cpp files. You may add any constants or permitted #includes you like to these files (e.g., StudentEditor.cpp may include header files like <list> or <set> and "StudentEditor.h", but not "StudentSpellcheck.h" or "StudentUndo.h"). You may also add support functions for these classes if you like (e.g., operator<). Make sure to properly comment your code.

For this project, you do not need to submit a report. You're welcome.

Good luck!