

PIC 10B Section 2 - Homework # 2 (due Sunday, January 17, at 11:59 pm)

You should upload each .cpp and .h file separately and submit them to CCLE before the due date/time! Your work will otherwise not be considered for grading. Do not submit a zipped up folder or any other type of file besides .h and .cpp.

Be sure you upload files with the precise name that you used in your editor environment otherwise there may be linker and other errors when your homeworks are compiled on a different machine. Also be sure your code compiles and runs on Visual Studio 2019.

WRITING A SPELL CHECKER

In this homework, you'll get to write a spell checking program. This will serve as a partial review of streams and some of the containers of the standard library (to be studied in-depth throughout this course).

Please refer to the syllabus for how the work will be graded: note that more than half of the marks come from good coding practices and code documentation.

You should write

- a header file **SpellCheck.h** that defines a **SpellCheck** class along with declaring its related functions/constructors,
- a file **SpellCheck.cpp** that implements all of the required functions/constructors, and
- a file **main.cpp** that contains a main routine that will allow the user to enter the name of a file **local to the source code folder** that will have its spelling checked, with asterisks surrounding each word that is found misspelled.

In the hints section, the logical sequence of events, practically pseudo-code, are provided. For a refresher on streams, see the 10A notes on streams. For a refresher on sets (unordered_sets have nearly the same functions) see the 10A notes on the Standard Library.

You can also find **dictionary.txt**, containing the set of valid spellings (**actually that file has a bunch of nonsense words I don't think are words but that's what happens when you download a random dictionary of words from the internet**), and **example.txt** (the example text where a spell check is done as a test case).

Just FYI: the story I included is a real story. This is a reason I'm not a huge skateboard fan... And don't even get me started on those darn scooter-bird-things :p

The basic functionality is that...

- The user is prompted for a file name to perform a spell check on. That **file will be stored in the same folder as the source code** and can be named anything at all (**do not** assume it will be called *example.txt*).
- For each word in that file, the spelling will be checked (this excludes any punctuation marks at the end and also converts a first-letter that is uppercase to a lowercase except for the word “I” that should not have its case changed).
- The entire input file, with all of its whitespace preserved, will be printed to the console, with asterisks * surrounding words that have been misspelled.
- We define the valid spelling of a word, other than “I”, as either the correct spelling all in lowercase letter or the correct spelling beginning with an uppercase letter but all other letters lowercase; for example, “Rice” and “rice” are both valid spellings, but “rICe” is not.

You must...

- write a **SpellCheck** class that
 - has a **dictionary** member variable of type **std::unordered_set**, found in the `<unordered_set>` header, to store the dictionary words (*use it just like a **std::set** but it is more appropriate in this context, which we’ll see when we study data structures*);
 - has a **default constructor** such that the class stores no words in its dictionary initially;
 - has a **read_dictionary** function that accepts the file name (of a dictionary text file) and reads all the words from that file into its dictionary of words;
 - has an **is_valid** function that accepts a word and returns **true** if the word is spelled correctly and otherwise **false**; and
 - has a **process_file** function that reads through a text file that needs to be spellchecked and prints the checked file to the console window.

To help with this implementation, you should also write free functions, i.e., non-member functions

- **is_white_space** that accepts a **char** and returns **true** if it is white space (a space, tab, or new line) and otherwise **false**;
- **final_punctuation** that accepts a word and returns a **bool** as to whether the final character is a punctuation mark; and
- **depunctuate** returning **void** that mutates a **std::string** input, making its first character lowercase and removing its final punctuation mark if there is one.

You may...

- assume the words will all be letters of the English language based on a standard US English language keyboard (it's actually a lot harder if you can't make this assumption, having to worry about locales and such!)
- assume the dictionary file will always be called **dictionary.txt** and will be stored in the same folder as the source code;
- assume the only possible punctuation marks that could appear (at the end of a word) are . , ! ? ; : with a - only appearing as part of a word;
- assume that there are no ellipses ...;
- assume there are no ‘ or “ used as quotations;
- assume there will never be two punctuation marks next to each other;
- assume the file will never end in white space (otherwise there are more conditions to check with the streams so this saves you some suffering)

Some initial guidance to get you on your way...

1. Be sure that you can read each word from the dictionary file, one word at a time. This will require working with file streams. Try reading each word from the file and printing it to the console (or maybe every 1000th word given how many words there are!).
2. Read over the member functions of **std::set** (an **std::unordered_set** behaves much the same). In particular, there is a **find** member function that returns an iterator and an **insert** function. Write code that can read every word from the dictionary file and store it in the unordered set.
3. Write code that allows a user to enter a word and determine whether or not that word is stored in the unordered_set.
4. Be sure that you can read from the dictionary file, store the words in an unordered_set of strings, and check if a word is in there before doing anything else!

The practical implementation for the **process_file** function can be summarized by:

- Start with an empty string (representing the console display at the end of the function call).
- While the stream is not exhausted:
 - While the next **char** in the stream is white space (use the function along with **peek** and **get** described below), add that white space to the display string one **char** at a time.
 - At this point the next **char** is not white space so read in the entire next word with **operator>>**. Find out if it is valid (use the function!).
 - * If the word is valid, add it to the display stream; otherwise
 - * format it appropriately by adding an * before its first char and either at the end or just before its terminating punctuation mark (use the function to check if it ends with punctuation) as appropriate and add the word to the display string.
- Then print that display string.

More help/hints:

If **c** is a **char**, the following lines of code will convert **c** to lowercase if it is uppercase. This works because **chars** are internally stored as integer types.

```
if ( ('A' <= c) && (c <= 'Z') ) {  
    c += ('a' - 'A');  
}
```

Input streams have member functions
// returns an int-value based on the next char and advanced the stream pointer
int get() const;
int peek() const; // returns an int-value based on the next char

If there are no more characters to extract from the stream, the return of these functions is **EOF**.

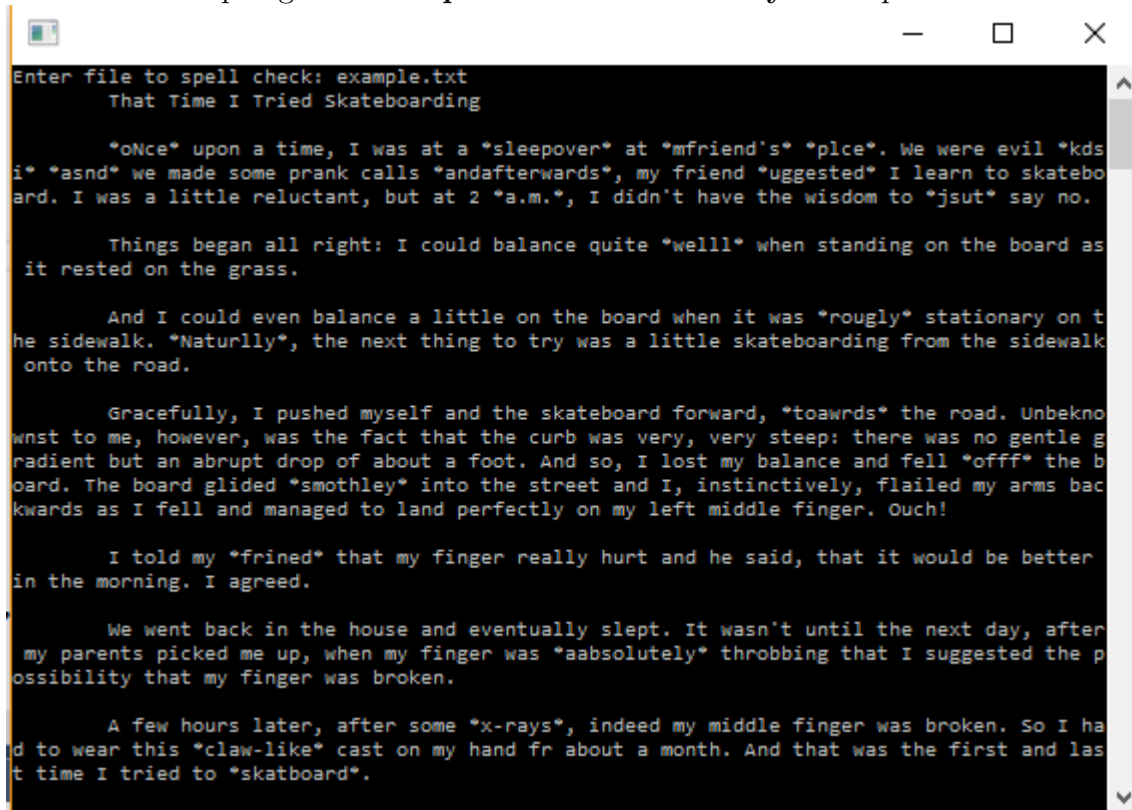
For example with **std::cin** (same idea holds for other input streams), if the stream held "abcde" with the stream pointer currently at **a** then:

```
// makes x into 'a' and advances the stream pointer to b  
char x = std::cin.get();
```

whereas if we now wrote

```
// makes y into 'b' and the stream pointer stays at b  
char y = std::cin.peek();
```

The desired output given **example.txt** with **dictionary.txt** is provided below.



```
Enter file to spell check: example.txt
That Time I Tried Skateboarding

    *oNce* upon a time, I was at a *sleepover* at *mfriend's* *plce*. We were evil *kds
i* *asnd* we made some prank calls *andafterwards*, my friend *uggested* I learn to skatebo
ard. I was a little reluctant, but at 2 *a.m.*, I didn't have the wisdom to *jsut* say no.

    Things began all right: I could balance quite *welll* when standing on the board as
it rested on the grass.

    And I could even balance a little on the board when it was *rougly* stationary on t
he sidewalk. *Naturllly*, the next thing to try was a little skateboarding from the sidewalk
onto the road.

    Gracefully, I pushed myself and the skateboard forward, *toawrds* the road. Unbekno
wnst to me, however, was the fact that the curb was very, very steep: there was no gentle g
radient but an abrupt drop of about a foot. And so, I lost my balance and fell *offf* the b
oard. The board glided *smothley* into the street and I, instinctively, flailed my arms bac
kwards as I fell and managed to land perfectly on my left middle finger. Ouch!

    I told my *frined* that my finger really hurt and he said, that it would be better
in the morning. I agreed.

    We went back in the house and eventually slept. It wasn't until the next day, after
my parents picked me up, when my finger was *aabsolutely* throbbing that I suggested the p
ossibility that my finger was broken.

    A few hours later, after some *x-rays*, indeed my middle finger was broken. So I ha
d to wear this *claw-like* cast on my hand fr about a month. And that was the first and las
t time I tried to *skatboard*.
```