




Lab 7 - FCC: Friendly Communication Checker

This lab is due on Monday, March 25th at 11:59pm.

FCC: Friendly Communication Checker


When **Dean Hachamovitch** 

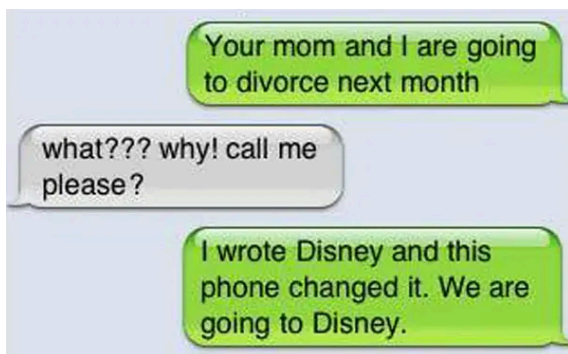
(<https://patentimages.storage.googleapis.com/94/57/79/8daff7714024e0/US6047300.pdf>) conceived of and patented the idea of a **"System and Method for Automatically Correcting A Misspelled Word"** , (<https://patents.google.com/patent/US6047300A/en>), there was no way that he could have imagined the way that it would change the world. Dean was the leader of the so-called Department of Stupid PC Tricks at Microsoft in the late 1990s when he developed autocorrect. After years working on the system, he advanced through the ranks at Microsoft and is now a vice president and head of data operations for the entire company. **[1]** 
(<https://www.wired.com/2014/07/history-of-autocorrect/>)

There is no doubt that we would be in a different place than today (where phones with virtual keyboards have become the standard) if it were not for autocorrect and predictive text entry (**T9** 



(<https://worldwide.espacenet.com/patent/search/family/024019996/publication/US5818437A?q=pn%3DUS5818437>), etc). If we had to type every word perfectly and correctly, we would likely still be using Blackberry devices and RIM would still be the smartphone manufacturing juggernaut it was in the late 1990s and early 2000s.

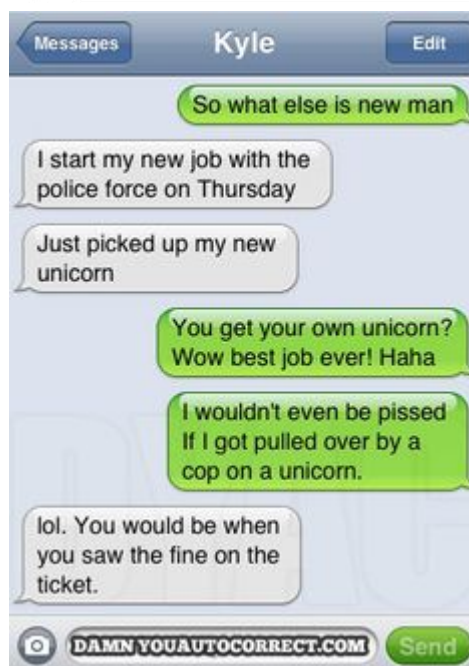
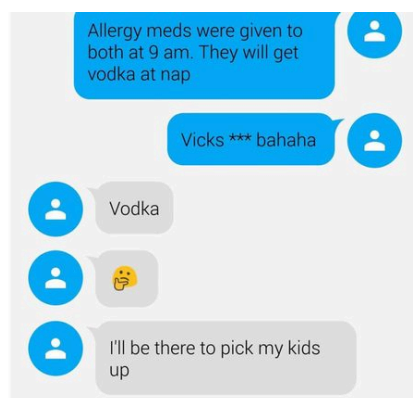
While there is every indication that our personal "productivity" has improved as a result of autocorrect, there are some downsides. For instance, the more often that autocorrect fails to read our minds, the more our frustration with the system grows. **[2]** , (<https://openreview.net/pdf?id=dcbsb4qTmnt>)



I'll probably recruit students in the fall to start in the spring, as long as I get Aton of work done before then

*a ton

(Really don't understand Apple autocorrect sometimes. WTH is Aton??)



What's more, autocorrect (and spellcheckers and grammar checkers more generally) may have had a negative impact on our ability to spell. A famous 2008 study by Stanford researchers found that spelling errors *increased* in student papers in the years between 1988 and 2008 -- a time

when spellcheckers became ubiquitous. [3] <https://www.jstor.org/stable/20457033> If you want to see how your (unaided) spelling stacks up against mine, play the MW "Spell-It" game [online](https://www.merriam-webster.com/word-games/spell-it) <https://www.merriam-webster.com/word-games/spell-it>. I scored 3440/4200.

Then there's the problem of relying too heavily on writing aids. Several researchers have shown that people who have especially high verbal skills (in other words, they are very good and communicating in writing) will make more grammatical mistakes when they write with a grammar checker enabled than otherwise. [4] <https://dl-acm-org.uc.idm.oclc.org/doi/10.1145/1070838.1070841>

And, finally, autocorrect may have hurt our skill as proofreaders and self-editors. Betsy DeVos, former secretary of the Department of Education famously tweeted an inspirational quote from W.E.B. DeBois. To bad his name is spelled DuBois. [4] https://www.mlive.com/news-us-world/2017/02/dept_education_spelling.html [Note: How many typos can you find in this lab writeup?]

This week you are going to get a chance to implement your own autocorrect system that fixes common typos. The Friendly Communication Checker (FCC) will improve your users' spelling and communication skills.

FCC will correct the spelling of a sentence (called the *fixme sentence*) read in from a file (call the *fixme file*). Of course, the FCC application is customizable and can be configured to handle precisely the words that a particular user has trouble spelling. The configuration is done in the *typos file* and the *fixos file*. The words in the *typos file* are the ones that the user has identified as hard to spell. The words in the *fixos file* are the same troublesome words, just spelled correctly!

For instance, I have a terrible time spelling restaruant and beatifual. The contents of typos/fixos file designed especially for me would look like

File	Contents
<i>typos file</i>	restaruant beatifual
<i>fixos file</i>	restaurant beautiful

Note: See how the positions of the words in the two configuration files are parallel. The X^{th} tricky word (from the left) in the typos file should be replaced in an autocorrected sentence with the X^{th} fixo word (again, from the left). There will be more discussion about the relationship between the configuration files (the *typos* and *fixos* files) and the input file (*fixme* file) below.

FCC requires a file (again, the *typos file*) that contains the list of words to fix (the *typos*) and a file (the *fixos file*) that contains their fixes (the *fixos*). The fixos in the fixos file and the typos in the typos file are separated by spaces. The first fixo in the fixos file is the corrected replacement for the first typo in the typos file. The second fixo in the fixos file is the corrected replacement for the second typo in the typos file (and so on).

FCC Vocabulary

<u>Term</u>	<u>Definition</u>
<i>fixme</i> <i>file</i>	The file containing the sentence to fix.
<i>fixo file</i>	The file containing all the fixos.
<i>fixo</i>	A replacement word for a typo.
<i>typo file</i>	The file containing all the typos.
<i>typo</i>	A misspelled or abbreviated word to be automatically corrected.

For the lab, ***you may assume*** that the sentences that FCC fixes will consist of

1. lowercase letters and numbers, and
2. will not contain any punctuation.

FCC will generate

1. a corrected sentence (i.e., the same sentence as the one read in from the *fixme* file but with the typos appropriately fixed) followed by
2. a period at the end; and
3. the number of autocorrections that it applied.

Yule luv this lab. Gud luck.

Program Design Task

As my dad always says, "There's only one opportunity to do something wrong -- might as well get it right." Before you begin, write out the pseudocode or draw a flow chart for your implementation of the FCC application.

Program Design Requirements

The pseudocode or flow chart that you write/draw to plan your implementation must describe the steps that you will take to

1. Read in the input from the fixos file and the typos file.
2. Correlate the lists of typos and fixos.
3. Guarantee that the list of typos and fixos have the same number of entries.
4. Read in the sentence to fix from the fixme file.
5. Iterate through the words in the sentence to fix and, for each,
 1. determine whether the word needs to be fixed; and
 2. if so, find and make the appropriate replacement.

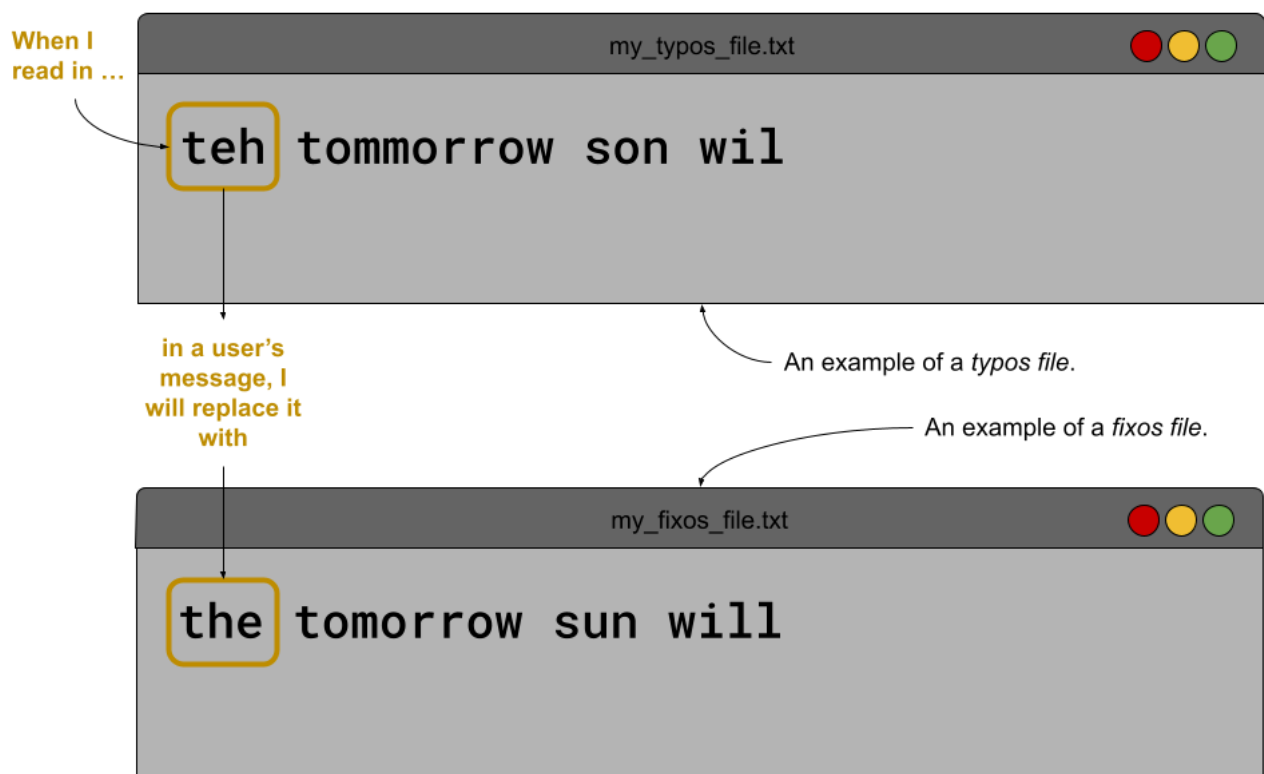
Programming Task

Each of the files used in this lab will consist of a series of words separated by spaces all on a single line.

If any one of the three files cannot be opened, that is an error. The sentence that the user wants to fix (which, again, is stored in the `fixme` file) may contain 0 or more words. There may be 0 or more typos configured to fix, depending on the user's preferences and history of typos.

The contents of the `fixos` file and the `typos` file are specifically designed so that it is possible to correlate words that are commonly misspelled with their proper spellings. The first fixo in the `fixos` file is the corrected replacement for the first typo in the `typos` file. The second fixo in the `fixos` file is the corrected replacement for the second typo in the `typos` file (and so on).

The Relationship Between the *Typos File* and the *Fixos File*



If the list of typos and the list of fixos is not the same length, that is an error.

Beginning with an empty *corrected sentence*, every word in the sentence to fix will be appended (unchanged) in the corrected sentence if it is correctly spelled. On the other hand, every one of the words in the sentence to fix that is a typo will be appended (in its corrected form) to the corrected sentence. In other words, FCC should perform a boolean calculation on each of the words in the sentence to fix: Is the word spelled correctly? If the answer to that boolean calculation is true, the FCC will append that word (verbatim) to the corrected sentence. Otherwise, if the answer to that boolean calculation is false, then FCC will append the corrected version of that word to the corrected sentence.

Determining the answer to the question of whether *the word is spelled correctly* can be done by comparing the possibly misspelled word with each of the typos. If the word possibly misspelled word matches (==) one of the typos, then it is misspelled.

The steps described in the two preceding paragraphs are the key to constructing a working FCC algorithm. I encourage you to read through the algorithmic description several times, write down (at least a first draft of) pseudocode and work through your design as you read the following example and complete the **practice sheet**

(<https://uc.instructure.com/courses/1657742/files/178316037?wrap=1>), (the practice sheet is also available in your skeleton).

As an example, suppose that the `fixme`, `typos` and `fixos` file have the following contents:

File	Contents
<code>fixme</code> file	teh son wil come out tommorrow
<code>typos</code> file	teh tommorrow son wil
<code>fixos</code> file	the tomorrow sun will

FCC will generate the following fixed sentence

the sun will come out tomorrow.

having made 4 corrections.

But, if the `fixme`, `typos` and `fixos` file have the following contents:

File	Contents
<code>fixme</code> file	teh son wil come out tommorrow
<code>typos</code> file	teh tommorrow son
<code>fixos</code> file	the tomorrow sun will

FCC will report an error (because the list of `fixos` and `typos` are not the same length).

FCC will also generate an error when the files have these contents

File	Contents
<code>fixme</code> file	teh son wil come out tommorrow
<code>typos</code> file	teh tommorrow son wil
<code>fixos</code> file	the tomorrow sun

because, again, the list of `fixos` and `typos` are not the same length.

Before beginning to either design or implement your solution, please complete the FCC Practice Sheet.

Programming Requirements

If you are a Windows-based C++ developer, begin with this **skeleton**

(<https://uc.instructure.com/courses/1657742/files/178316021?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/178316021/download?download_frd=1) . If you are a macOS-based developer, begin with this **skeleton**

(<https://uc.instructure.com/courses/1657742/files/178315975?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/178315975/download?download_frd=1) . If you are a Linux-based developer, start with this **skeleton**

(<https://uc.instructure.com/courses/1657742/files/178316391?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/178316391/download?download_frd=1) . You must use the skeleton code to successfully complete this lab.

All of the code that you write for FCC will go in the `fcc.cpp` file. In particular, your job is to implement the `fcc` function. A stub implementation of the `fcc` function is included in the skeleton. The `main.cpp` file contains the `main` function for the FCC program and executes a series of unit tests to check that your implementation of the `fcc` function meets all the requirements. Before you make any changes, the code in the skeleton will compile and run but will not pass the unit tests.

Read and study the declaration of the `fcc` function. Which parameters are passed by reference? Which parameters are passed by value? What is the return type of the function?

The value of the `fixme_filename` parameter will be the name of the file that contains the sentence to fix. The value of the `typo_filename` parameter will be the name of the file that contains the typos. The value of the `fixo_filename` parameter will be the name of the file that contains the fixos for the typos. During the execution of the `fcc` function, you will set the value of the `fixed_sentence` function based on the words, typos and fixos in the files named by the `fixme_filename`, `typo_filename`, and `fixo_filename` parameters, respectively.

If it were me, I would start by adding some debugging code to the body of the `fcc` function that simply prints out the values of the parameters every time the function is called. I would then run the program and examine the output. Why do this extra step? Does it help you understand the purpose of the code you have to write? Do the outputs correspond to any files given to you in the skeleton? I would answer those questions before going further.

The return value of the `fcc` function plays several roles. In the case where all the input parameters are properly specified and the function executes without error (see above for the two situations that constitute an error), the return value will specify how many words in the file specified by the `fixme_filename` parameter were autocorrected.

When there is an error in the execution of the `fcc` function (see above for the conditions under which an error occurs), the `fcc` function will return `-1`.

The `main` function in `main.cpp` contains a series of unit tests. When your `fcc` meets the specifications of the lab, you will see an appropriately pleasant message printed on the screen. When the unit tests detect an error, a descriptive message will be displayed. Read that message closely for information about where to start debugging.

In order to iterate through a space-separated list of strings in a file, you will resort to a loop. To make it easier to control when to stop reading from a file (i.e., to detect when the file has no more data), the skeleton code contains a function named `more_to_read`. The function takes a `std::ifstream` as a parameter (by reference!) and returns a `bool` ean. The value returned by a call to the `more_to_read` function is `true` if there is, well, more data to read from the file and `false` if the file has been completely read. I **highly recommend** that you use this function in your solution.

In the body of your implementation of the `fcc` function, you will probably end up with a loop structure that looks like the following:

```
declare a fixos vector
declare a typos vector
while (data can be read from the fixos file) {
    read the next word from the fixos file
    store that word in a fixos vector
}
while (data can be read from the typos file) {
    read the next word from the typos file
    store that word in a typos vector
}
```

Each of those loops will cease when the files run out of data to be read.

When the loop terminates, you are in a position to check whether any errors occurred while reading the input data. Think about the loops' *postconditions* ("... a statement of the conditions at the end of a" block of code (F. M. Carrano and T. D. Henry, *Data abstraction & problem solving with C++: Walls and mirrors*, 7th ed. Upper Saddle River, NJ: Pearson, 2016.)) that you could check to make sure that the number of typos and the number of fixos matches. See below for additional documentation on using `std::vector`.

Programmers are not forced to write functions -- they want to write functions. Why? Because functions allow programmers to hive away functionality behind a curtain and use that functionality to easily implement a piece of an algorithm. Once a programmer knows that a function works, they are free to rely on it and do not need to expend the mental energy considering it's mechanics while they move on to develop other parts of the system. It's the same thing in real life -- how much more mental space do we have to ponder the allegiances in Game of Thrones when we don't also have to think about all the functionality required for the TV to display an image.

In this lab you are not directed to write any particular functions to help you implement the `fcc` function. However, it will be something that you *want* to do. To incentivize you to take advantage of the power of abstraction, you are required to implement *at least* two helper functions for use in the implementation of the `fcc` function. You must document those functions according to the following specification:

```
/*
 * <function name>
 *
 * <short description of what the function does>
 *
 * input: <short description of all input parameters>
 * output: <short description of all output parameters>
```



```
*  
* and the return value>  
*/
```

Tips and Tricks:

If you use the stream extraction operator to read into a `std::string`-typed variable from an input stream (remember, that could be either `std::cin` or an input stream associated with a file (e.g., a `std::ifstream`)), the operation will assign all the characters between the stream's readhead and the next separator (usually a space!) to the `std::string`-typed variable. This fact might be useful for your implementation of the lab.

As an example, consider the following program

```
#include <iostream>  
#include <vector>  
  
int main() {  
    std::vector<std::string> words_in_sentence{};  
    std::string next_string{};  
  
    while (std::cin >> next_string) {  
        words_in_sentence.push_back(next_string);  
    }  
  
    for (auto str : words_in_sentence) {  
        std::cout << "str: #" << str << "#\n";  
    }  
}
```

When this program is executed with input on the keyboard like

one plus 1 is two.

will print

```
str: #one#  
str: #plus#  
str: #1#  
str: #is#  
str: #two.#
```

as output (see that code live online [here](https://godbolt.org/z/Wc68xzf6M)  (https://godbolt.org/z/Wc68xzf6M)).

As a way to get started, let's walk through the code (above), study what it does, adapt it to fit our precise needs and tick off one of the requirements of this lab by writing a useful helper function.

Scroll back up and look for anything that seems awkward about our pseudocode for reading the data from the fixos and typos files. Before trying to figure out how we could translate that pseudocode to actual C++, let's make sure that we know *why* we want to write something like that.

As you know, the contents of the typos/fixos file are related. To understand that relationship, let's give each word in the typos/fixos file an *index* depending upon its position relative to the other words in the file (and let's count starting at 0 and work left to right).


As an(other) example, let's look at what might happen if the contents of the configuration files (typos and fixos) looks like

File	Contents
typos file	nuthin shur
fixos file	nothing sure

nuthin would have index 0. shur would have index 1. nothing would have index 0. sure would have index 1. See the parallelism?

What would happen if I was attempting to determine whether or not I had to replace "shur"? My first job would be to *iterate* through the words that are in the typos file and look for a match for "shur". Sure enough, there it is ... "shur" is the *second* word in the list of words in the typos file (with a 1 index!). So, to make FCC work as it is supposed to, I would make sure that I replaced "shur" with ... I'll wait ... the second word (with a 1 index) in the fixos file.

Pretty straightforward and really neat how it is possible to use *parallel* areas to build this powerful software.

To pull out different words from an input file according to their index, we would have to write code that moves the readhead back and forth in an input stream (again, either the keyboard or a file). That's not an **impossible**  (https://en.cppreference.com/w/cpp/io/basic_istream/seekg) task, but it's also not fun! Good thing that there is an alternate solution!

What have we learned in class that are designed *exactly* to give us random access to individual items from a group of elements (which all have the same type)? That's right, a vector! Our coding will be much, *much* more straightforward if we are able to *parse* the list of words in the fixos and typos file into a vector! Once we have pulled the words out of the file and put them into a vector, we can reach in and get individual words with relative ease.

Now, go back one more time and take a look at the pseudocode that we wrote and the sample C++ code that I showed. Notice that the code seems to accomplish most of what we need to translate our pseudocode into working code.

First, let's make our declarations:

```
declare a fixos vector  
declare a typos vector
```

Perfect, we can translate that to

```
std::vector<std::string> fixos;  
std::vector<std::string> typos
```

Okay, now we can work on our while loop:

```
while (data can be read from the fixos file) {
```

If we only had a way to determine whether there was still data available to read from a file ... oh wait, we do: `more_to_read`.

```
while (more_to_read(fixos_file)) {
```

How cool! And now, what about ...

```
read the next word from the fixos file
```

Well, as we learned from seeing the code above, we know that we can read into a `std::string`-typed variable using the stream extraction operator and it will read all pieces of data until there a whitespace. That sounds like exactly what will do the trick to "read the next work from the fixos file":

```
std::string next_word;  
fixos_file >> next_word
```

And, for the last step

```
store that word in a fixos vector
```

I think that `push_back` will do perfectly:

```
fixos.push_back(next_word)
```

Voila! We are done:

```
std::vector<std::string> fixos;  
std::vector<std::string> typos;  
while (more_to_read(fixos_file)) {  
    std::string next_word;  
    fixos_file >> next_word;  
    fixos.push_back(next_word);  
}
```

Before we go too far, let's not forget that we will have to do the same thing for the typos. Ugh. Copy and paste, here we come!

```
while (more_to_read(typos_file)) {  
    std::string next_word;  
    typos_file >> next_word;
```

```
    typos.push_back(next_word);  
}
```

After all that work coding, I've forgotten just *why* we wanted to read the words from the file and put them into a vector. Oh yes, I remember now ... we wanted to pull out words from a particular place (maybe some index `i` in a loop that we will write in the future) in the list:

```
fixos[i]
```

Awesome.

We've written lots of code and not yet determined that it will work. Whenever I am coding, I try to write as little code as possible before making sure that it runs and does *something*. As we know from our earlier experiments, our fcc function gets invoked by code in the main function. Each time it is invoked, it will have different values for the parameters. If it were me, I would test what we have written so far by adding some additional code to what we just wrote.

What exactly would I add? Well, think about two variables that we used in the code above that are not yet defined. We will need code to declare `fixos_file` and `typos_file`. Declaring those files is not enough, however. We will also need to associate those `std::ifstream`-typed variables with files on the user's computer. Make those additions and then come back to me!

Welcome back. If we run the code that we have now, we aren't going to get any output that will help us determine whether our code works or not. Again, if it were me, I would add some debugging code after the two loops that we have already written. I would add two *more* loops: one to go through the contents of the `fixos` `std::vector` and one to go through the contents of the `typos` `std::vector`. Those loops would print out each of the values in the `std::vectors`. I would then determine whether or not all the values that should be in the vectors are.

Once I have determined that my loops work and fill the two vectors with the correct values each time fcc is called, I know that I am firmly in the driver's seat of a Maseraty speeding toward success.

Pothole! There's a problem ... or should I say two problems. Can you see them? Of course you can. We have duplicated code. It is just begging for refactorization into a ... function! Let's make a function that reads each of the words from a file and puts them into a vector.

What do we need in order to read data from a file? Well, a file (`std::ifstream`), obviously. So, that'll be our parameter type. Because we are going to be reading from the file passed as an argument, we are going to need to make modifications to it *and* make it so that the caller can see any of those modifications. That means that we will need to have that as a by-reference parameter.

Now, what will the function return? Yes! We will return the newly created vector that contains the words from the given file. So, the return type of the function will be `std::vector<std::string>`. The

last thing that we need to determine is the function's name. I will leave that up to you, but I think that I will name mine `read_from_file`. So, here's how we can start our refactorization:

```
std::vector<std::string> read_from_file(std::ifstream &file) {  
}
```

Yes! Now, let's just bring up the code that should go into the body:

```
std::vector<std::string> read_from_file(std::ifstream &file) {  
    std::vector<std::string> typos;  
    while (more_to_read(fixos_file)) {  
        std::string next_word;  
        fixos_file >> next_word;  
        fixos.push_back(next_word)  
    }  
}
```

Okay, well, we need to make some clean ups. `typos` is not a very descriptive name anymore. In its current use, the vector we are populating could refer to either the typos or the fixos, depending on the argument. So, we'll make that more generic:

```
std::vector<std::string> read_from_file(std::ifstream &file) {  
    std::vector<std::string> words;  
    while (more_to_read(fixos_file)) {  
        std::string next_word;  
        fixos_file >> next_word;  
        words.push_back(next_word)  
    }  
}
```

And, let's fix that syntax error of using `fixos_file` rather than `file`:

```
std::vector<std::string> read_from_file(std::ifstream &file) {  
    std::vector<std::string> words;  
    while (more_to_read(file)) {  
        std::string next_word;  
        file >> next_word;  
        words.push_back(next_word)  
    }  
}
```





Now *that* is some really good looking code! We are only missing one final thing ... we need to tell C++ that words is the result of our function:

```
std::vector<std::string> read_from_file(std::ifstream &file) {
    std::vector<std::string> words;
    while (more_to_read(file)) {
        std::string next_word;
        file >> next_word;
        words.push_back(next_word);
    }
    return words;
}
```


After replacing the loops in our fcc function with invocations of the function we just wrote, our testing/debugging code should still work as it did before! Check to make sure that is the case!

With that awesome function written, you are well on your way (90%, I would say) to a nicely constructed code base for the FCC application!

Critical Thinking Task:


Autocorrect (and assistive technology in general) is an example of the influence of computer science in society. Today, modern life is increasingly defined by applications built on machine learning and *big data*. The Internet defines big data as "a field that treats ways to analyze, systematically extract information from, or otherwise deal with data sets that are too large or complex to be dealt with by traditional data-processing application software." Computer scientists write algorithms to make sense of these data sets. The results of their analysis are used throughout society: **[predictive policing](https://www.washingtonpost.com/opinions/big-data-may-be-reinforcing-racial-bias-in-the-criminal-justice-system/2017/02/10/d63de518-ee3a-11e6-9973-c5efb7ccfb0d_story.html)**  (https://www.washingtonpost.com/opinions/big-data-may-be-reinforcing-racial-bias-in-the-criminal-justice-system/2017/02/10/d63de518-ee3a-11e6-9973-c5efb7ccfb0d_story.html), **[availability of credit](https://www.bostonfed.org/publications/research-department-working-paper/2019/how-magic-a-bullet-is-machine-learning-for-credit-analysis.aspx)**  (<https://www.bostonfed.org/publications/research-department-working-paper/2019/how-magic-a-bullet-is-machine-learning-for-credit-analysis.aspx>), **[self-driving cars](https://bdtechtalks.com/2020/07/29/self-driving-tesla-car-deep-learning/)**  (<https://bdtechtalks.com/2020/07/29/self-driving-tesla-car-deep-learning/>), and, of course, **[advertising](https://www.ibm.com/watson-advertising/thought-leadership/benefits-of-machine-learning-in-advertising)**  (<https://www.ibm.com/watson-advertising/thought-leadership/benefits-of-machine-learning-in-advertising>).


The Utopian vision of universal societal improvement through the application of algorithms to data is clouded by the fact that engineers and developers a) choose the data they analyse and b) write the methods of analysis. These two facts mean that developers have a tremendous responsibility to make sure that their analyses are accurate and equitable.

Many groups/individuals are studying ways to make sure that the application of algorithms to big data does not disproportionately harm particular groups of people. Your task is to find and document such a group/individual and summarize their work. 

Critical Thinking Requirement:

In 500 words or less, describe a group/individual who is studying ways to make sure that the application of algorithms to big data does not disproportionately harm particular groups of people. All references to external resources must be properly documented and formatted. The choice of

formatting for external references is up to you, but you may find it helpful to consult the Purdue OWL for **information** .

(https://owl.purdue.edu/owl/research_and_citation/apa_style/apa_style_introduction.html). The Purdue OWL also has extensive information on ways to **avoid plagiarism**  (https://owl.purdue.edu/owl/avoiding_plagiarism/index.html).

Deliverables:

1. In PDF format, the pseudocode describing the algorithms you used to implement the `fcc` function (and its (at least) two supporting functions) (named `design.pdf`).
2. In PDF format, the completed FCC Practice Sheet (named `fcc-practice.pdf`)
3. The C++ source code containing the implementation of the `fcc` function required for successful completion of the *FCC* application (named `fcc.cpp`).
4. The response to the Critical Thinking prompts in PDF format (named `m1.pdf`).

Rubric:

Points	Task	Criteria
15	Design	Pseudocode uses algorithmic thinking to express a high-level description of how to implement the <i>FCC</i> autocorrect application by meeting the criteria specified in the Program Design Requirements section.
10	Design	Your FCC Practice Sheet is completed accurately.
15	Critical Thinking	In 500 words or less, your response to the Critical Thinking task describes a group/individual who is studying ways to make sure that the application of algorithms to big data does not disproportionately harm particular groups of people. Your references to external resources are properly documented and formatted.
10	Programming	Each function you wrote (especially <code>fcc</code>), is properly documented with comments that match the format given in the Programming Requirements section.
25	Programming	<i>FCC</i> application behaves correctly on sample data included with the skeleton code for this lab.
15	Programming	<i>FCC</i> application behaves correctly on other test inputs
10	Programming	Code for each function you wrote matches the pseudocode and all variables are given appropriate/meaningful names.

Related Learning Objectives:

1. Writing boolean expressions using relational and logical operators
2. Using if-statements to implement selective program execution
3. Using algorithmic thinking to design programs
4. Writing syntactically correct for/while loops
5. Using/accessing/manipulating vectors

6. Using and manipulating parallel arrays to model data relationships
7. Using methods on objects