CSCI-1200 Data Structures — Fall 2019
Homework 1 — Text Justification

Before starting this homework, make sure you have read and understood the "Academic Integrity Policy".

This homework assignment is a programming exercise involving simple text manipulation to format input text into a framed column with a specified width. You will work with command line arguments, file input and output, and the C++ Standard Template Library `string` and `vector` classes. Please read the entire assignment before starting to program. When you are ready to begin, create a subdirectory for homeworks within your main Data Structures course directory and within that directory create a subdirectory `hw1` for this assignment.

Your program will read words from an input file and greedily pack the words one at a time onto each line, making sure *not to exceed the specified width.* Your program will support three different methods for dealing with any extra, unused space on each line. The first, *flush left*, simply positions each word as far to the left as possible, leaving a gap of spaces (if any) on the right. The second, *flush right*, does the opposite, pushing all the words to the right, and puts the extra spaces on the left. In the final mode, *full justify*, the extra spaces are distributed evenly between the words. If the number of extra spaces on the line do not divide evenly into those "between" slots, the slots on the left will be assigned more spaces that the slots on the right.

Let's look at a simple example. Here is a sample input file, named `example.txt`:

```
Here is an example of text justification.
```

And here's the output produced for that input file for the three different justification modes, for the specified text width of 16:

flush left

```
--------------------
| Here is an       |
| example of text  |
| justification.   |
--------------------
```

flush right

```
--------------------
|       Here is an |
|  example of text |
|   justification. |
--------------------
```

full justify

```
--------------------
| Here    is    an |
| example  of text |
| justification.   |
--------------------
```

First note that the output includes a simple ascii art line box drawn around the text, but those characters do *not* count towards the specified line width of the text. In other words, each line of final output is actually 4 characters longer, in this example, 20 total characters. Notice that the same words appear on each line in all three cases. The only difference is where the "extra" spaces appear. The top line has 6 extra spaces (16 - # of characters in "`Here is an`" = 6). In full justify mode, these spaces are evenly divided into the 2 slots between the 3 words. The second line has 1 extra space (16 - # of characters in "`example of text`" = 1). This extra space is assigned to the leftmost of the two slots on the second line. Also note a typical convention for full justify formatting: The last line of a block of text formatted with full justify is *not* forced to stretch all the way to the right edge. No extra space is inserted between words on the last line – it should be flush left justified only.

File I/O

You will read the words to format from an input text file. You should not make any assumptions about the formatting of this file, except that the words will be separated by at least one *whitespace character*. Remember that whitespace characters include spaces, tabs, and newlines. Any punctuation in the file (including periods, commas, apostrophes, etc.) should be treated as part of the word if it is not separated from the word by spaces. The basic iostream string input operator, `>>`, will work perfectly for this assignment. The output of your program will be written to a file, and should follow the specifications in this handout and match our examples. Reading and writing files in C++ is very similar to `std::cin` and `std::cout`. See examples of STL file streams on the course webpage "Helpful C++ Programming Information".

Command Line Arguments

Your program will expect 4 *command line arguments*. The first is the name of the input file. The second is the name of the output file. The third argument is an integer that specifies the width of the text column. The fourth argument will be a string (`flush_left`, `flush_right`, or `full_justify`) specifying which formatting mode should be used. Here are examples of valid command lines for your program:

```
./justify.exe example.txt example_16_flush_left.txt 16 flush_left
./justify.exe example.txt example_16_flush_right.txt 16 flush_right
./justify.exe example.txt example_16_full_justify.txt 16 full_justify
```

You should implement very simple error checking to ensure that 4 arguments are provided and that the input and output file streams are successfully opened. You should also check that the values for the third and fourth arguments are valid. Your program should exit gracefully with a useful error message sent to `std::cerr` if there is a problem with the arguments.

You must follow the specifications for the command line, input file, and output file **exactly** to ensure you receive full credit from the Submitty homework submission autograder. We have provided sample input & output files on the course website. Examples of using command line arguments can be found on the course webpage: "Helpful C++ Programming Information".

Corner Cases

So now you understand the core requirements for this assignment, but you may already have thought of a few problem cases. For example, how should the program full justify a line that contains just a single word? There are no available slots between words to use to insert the extra spaces. In this case your program should simply left justify the line. Another question you might ask is what to do if one of the words in the file is (by itself!) wider than the width of the column? In your initial coding and testing we recommend you assume that this will never happen. This solution will be worth nearly full credit. To receive full credit on the assignment your program should handle this case by splitting the word and inserting a hyphen. Note that we do not expect you to *properly* split the words between syllables, as that would require a database of English words and syllables. If you think of other corner cases as you work on the assignment, propose reasonable ways to handle those situations. If those solutions are overly complex or tricky, you do not need to tackle the implementation, but you should write up your thoughts in your README.txt for the grader to read.

Extra Credit

Once the fundamental requirements of the assignment are completed, you may implement some fun extensions to earn a few points of extra credit. Rather than formatting the text inside a simple rectangular column, output the text inside of a triangular, trapezoidal, or even circular frame. Document your extensions in your `README.txt` file and include a sample command line and sample output file with the results.

Submission Details

Use good coding style when you design and implement your program. Organize your program into functions: don't put *all* the code in `main`! Be sure to read the "Good Programming Practices" as you put the finishing touches on your solution. Be sure to make up new test cases to fully debug your program and don't forget to comment your code! Use the provided template `README.txt` file for notes you want the grader to read. You must do this assignment on your own, as described in the "Collaboration Policy & Academic Integrity" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file. Prepare and submit your assignment as instructed on the course webpage. Please ask a TA if you need help preparing your assignment for submission or if you have difficulty writing portable code.