Project 5 Report

1. The first obstacle I overcame was dealing with skipped words or characters in my output. At first I used while(inf.get(c)) twice, once in my main function and again in my separate word-filling function. This caused every word to lack its first letter, so I adjusted to use while(inf.get(c)) once, within my secondary word-filling function. I changed my while loop into an empty for (;;) loop within my main function.
2. While testing my function on different inputs, I had a lot of errors with spaces at the start of lines. I first added cout statements to each branch of my function to identify the source of incorrect output.
3. I struggled to place spaces after punctuation, since pasting spaces after every word that ended in a period or question mark often created excess spaces. I realized that I needed to work backwards, by placing an extra space in front of words FOLLOWING those with punctuation. I did this by creating a tracker integer variable that = 1 whenever the previous word ended in punctuation and was reset to 0 if the function ran multiple times and did not use it.
4. I struggled a lot with preventing my function from outputting newlines at the end of my output when the end of the input contained consecutive #P#. I had to my program to add newlines individually to each valid condition, rather than having a single branch that outputted two newlines every time it encountered a non-consecutive #P#.
5. The final obstacle I overcame was making sure that consecutive #P# in the input didn’t cause multiple blank lines. To do this, I added an “if” statement that checked if an int wordCount > 0. Each time that this branch runs, it resets wordCount to 0. Therefore, if #P# occurs multiple times in a row, its first occurrence reset wordCount and prevents extra empty lines.

**Overall Routine Description**

My routine uses a pair of functions to receive and format input from a file into another file. My secondary function, fillWord, uses a while statement to receive input character-by-character and fill a character array until it hits whitespace. It then returns an integer to my main routine to tell it that it has successfully filled a word, and my main routine uses a long if-else tree of conditions to determine how many spaces and newlines to insert before and after these words.

**Branch-by-branch Pseudocode:**

[Main body of routine: these branches receive single words from my secondary function fillWord, and check these words against if-else conditions to format them]

*if fillWord successfully fills a character array with a single word from the input file:*

***if the word is #P#***

*{*

*isParagraph = true;*

*// the next word will use this info to determine whether or not to add an extra newline*

*}*

***else if the word is either the first word of input OR following a word that filled a whole line***

*{*

*if the word is following a #P#*

*output a newline;*

*output the word;*

*set the character count equal to the number of characters in the word, so later words can account for line space;*

*}*

***else if the word is longer than the line length***

*{*

*set bool LONG\_WORD = true;*

*// for later use to return 1*

*if the word is NOT the first input word and the previous word did not fill its entire line*

*output the appropriate number of newlines;*

*for (as many times as the line length fits into the word)*

*{*

*fill character array longWord[] with lineLength*

*characters;*

*output longWord;*

*}*

*fill character array longWord[] with any leftover letters that weren’t used in previous lines;*

*output longWord;*

*if the word was an exact multiple of lineLength*

*{*

*set wordCount to 100 so the next word can account for the extra newline;*

*set charCount to 0 because the next word is starting on an empty line;*

*}*

*else*

*{*

*set wordCount to 1 because the line now has one word;*

*set charCount to the length of this word;*

*}*

*}*

***else if the previous word ended in '.' or '?' and this word will fit on the current line***

*{*

*output two spaces;*

*output the word;*

*increase wordCount;*

*}*

***else if the previous word ended in '.' or '?' and this word will NOT fit on the current line***

*{*

*output a newline;*

*output the word;*

*set wordCount to 1;*

*set charCount to the length of the word;*

*}*

***else if this word has no special conditions and it will fit on the current line***

*{*

*if the previous word was #P#*

*{*

*output an empty line;*

*output the word;*

*set charCount to the length of this word;*

*set wordCount to 1;*

*}*

*else*

*{*

*output a single space;*

*set charCount to the length of this word and the space;*

*increment wordCount;*

*}*

*}*

***else if this word will NOT fit on the current line***

*{*

*if the previous word was #P#*

*{*

*output an empty line;*

*output the word;*

*}*

*else*

*{*

*output a newline;*

*output the word;*

*}*

*set charCount to the length of this word;*

*set wordCount to 1;*

*}*

***/// THESE IF-ELSE STATEMENTS WORK AS INDICATORS SO THE NEXT WORD KNOWS WHETHER***

***THE PREVIOUS WORD WAS A SPECIAL CASE ////***

***if the word is #P#***

*storeParagraph = 1 ; // integer so the next word can check if the previous word was a #P#*

*isParagraph = false; // resets isParagraph so each #P# is only accounted for once*

*else*

*storeParagraph = 0; // resets isParagraph so each #P# is only accounted for once*

***if the word ends in ‘.’ or ‘?’***

*{*

*storePunctuation = 1; // integer so the next word can check if the previous word ended in ‘.’ or ‘?’*

*hasPunctuation =* ***false****; // resets hasPunctuation so each ‘.’ or ‘?’ is only accounted for once*

*}*

***else***

*storePunctuation = 0; // resets hasPunctuation so each ‘.’ or ‘?’ is only accounted for once*

*}*

**TEST DATA**

|  |  |
| --- | --- |
| Test Data | Reason |
| fill(0, inf, outf) | the desired maximum line length is less than 1 |
| fill(351, inf, outf) | If the first parameter of fill is greater than 350, return 2 without writing any output |
| fill(9, inf, outf)  input:  1234567891 | input word is longer than the maximum line length |
| fill(9, inf, outf)  input:  123456789123456789123 | Input word is over twice the maximum line length. The rest of that word must begin the next output line, subject to similar splitting when it’s too long. |
| fill(9, inf, outf)  input:  123456789 hello there? | Input word is the same length as the maximum line length |
| fill(10, inf, outf)  input:  1st line 2nd line | Fit as many words in an output line without exceeding line length |
| fill(10, inf, outf)  input:  Hello how are you  Doing today? | Words in an output line must normally be separated by one blank. |
| fill(10, inf, outf)  input:  This. Has? Two. Spaces | Two spaces after a ‘.’ and ‘?’ |
| fill(9, inf, outf)  input:  123. 789 1234. 789 | The two spaces after a ‘.’ or ‘?’ are accounted for in terms of character limit when fitting a word onto a line |
| fill(10, inf, outf)  input:  .This. ?Has? T.wo. Spaces | A period or question mark anywhere other than the last character of a word should not have any effect on spacing |
| fill(10, inf, outf)  input:  End here. | Confirming that there will not be extra spaces after the final output when there input ends in a period |
| fill(10, inf, outf)  input:  End here? | Confirming that there will not be extra spaces after the final output when there input ends in an question mark |
| fill(10, inf, outf)  input:  End here? | Confirming that there will not be extra spaces after the final output when there input ends in an question mark and newlines and spaces |
| fill(10, inf, outf)  input:  Hello there. #P# | there must not be empty output lines after the last word of the last paragraph. |
| fill(10, inf, outf)  input:  123 #P# 123 #P# 123. | The input word #P# is not to be processed as a word according to the above rules; instead, it indicates a paragraph break. |
| fill(9, inf, outf)  input:  123456789 #P# 123456789 #P# 123 #P# 12345 | #P# will add an empty line if it comes after a word that is the same length as lineLength. |
| fill(9, inf, outf)  input:  #P# #P# #P# 123 #P# 123 #P# #P# 123 #P# | The very first output paragraph must not be preceded by an empty line.  The very last output paragraph must not be followed by an empty line.  Two or more consecutive #P# words in the input are treated as just one paragraph break |
| fill(9, inf, outf)  input:  banjo#P# hello there? | banjo#P# is one eight-character word; it does not cause a paragraph break, because in that string, #P# is not a word because of the immediately preceding non-whitespace character o. |

|  |  |
| --- | --- |
| fill(9, inf, outf)  INPUT:  #P#  12 456 89 1 3 5 7 9 123456789 1234567891 3 5 7 9 #P#  12 #P# #P# #P#  123456789 123 567. 123456789 #P#  1. 5? 9 12345678912345678912345. 9  END HERE.  #P# #P# #P# | * No empty lines before first output word * Words will fill a line until it hits the line length * A word equal to the line length will begin on its own line (and the previous line will not have any residual spaces) * A word longer than the line length and following a word that was equal to the line length will begin on its own line AND break off appropriate into the next line * The words following a line that was longer than line length will begin on the same line as the excess * A paragraph break will create an empty line * A word will be on its own if it is surrounded by paragraph breaks on both sides * Consecutive paragraph breaks will create only one empty line * A word equal to the line length will begin on the correct line following a paragraph break * Words will fill a line until it hits the line length, accounting for the number of spaces needed after a period or question mark * A word equal to the line length will start on the appropriate line following a full line * A paragraph break following a word equal to the line length will create an empty line * Spaces created by punctuation are accounted for when filling to line length * A word multiple times as long as lineLength will break apart multiple times in the correct places * A single-character word will fit on a line if there is only one character space left before it is full * Output will end with a single newline as the final character after the last word of input * Multiple consecutive #P# at the end of input will not create empty lines after the last output word |
|  |  |