**Obstacles**

There were two main obstacles that I overcame in the development of Project 3. The first was that I misunderstood how string.substr() worked in C++. Because Java created the substring from index1 to index2, I assumed that the result would be the same. However, in C++, it creates a substring from index1 for a certain length.

My second obstacle was keeping the party result as 2 digits or less. I had to create flags that would count the number of digits in a row, and return that the syntax was invalid if there were 3 or more in a row.

**Description**

My function for *hasProperSyntax* is designed based off of dividing strings into chunks. For the function, I created another helper function, bool *checksSyntaxForOne(string oneState).* *hasProperSyntax* first checks whether or not a string is empty. Then, it counts the number of commas in *pollData* as a the variable *countCommas.* It creates another string called data equal to *pollData*. It creates a while loop according to this pseudocode:

repeatedly  
 make a string called *one* from 0 to the next comma in pollData

if *one* is invalid using *checksSyntaxForOne*

return false

otherwise

decrement *countCommas*

data becomes a substring from the index of comma onwards

return true

This checks *countCommas* number of chunks+1, which will check all parts of the string pollData.

Within *checksSyntaxForOne(string oneState)*, it performs several different checks. oneState is a chunk between commas. If the size is 0, that means there are multiple commas after each other, which is invalid and thus returns false. Then, it creates a string that is the first two characters of *oneState* called *stateCode*. If *isValidUppercaseStateCode(stateCode*) is false, the chunk *oneState* must be invalid, so it returns false. Then, I created a string called *rest* that is everything but the *stateCode*. I created two variables, *numbersInRow* and *numLettersInRow* to keep track of how many numbers and how many letters are in a row. Because there cannot be 3 digit party results or 2 letters after one another, it returns false given those conditions. It creates a for loop that iterates through *rest* according to this pseudocode:

repeatedly

if it’s a digit

set *numLettersInRow* to 0

increment *numbersInRow*

if *numbersInRow* is greater than 2

return false

if it’s at the end of the string

return false

if it’s D,d,I,i,R, or r

set *numbersInRow* to 0

increment *numLettersInRow*

if *numLettersInRow* is greater than 1

return false

otherwise

return false

return true

Finally, *tallySeats(string pollData, char party, int& seatTally)*. First, I said that if *hasProperSyntax(pollData)* was false, it should return 1. Otherwise if the party wasn’t alphanumeric, return 2. I made an integer *seats* to keep track of the amount of seats. Otherwise, it ran through the *pollData* string to count the party result using this pseudocode:  
 repeatedly:

if it finds a party character

repeatedly until a nondigit is found

increment *numBack*

get a substring from the index of the nondigit+1 *numBack* amount of times

convert that substring to an integer

if that integer is greater than 99, set it to 0

increase *seats* by the integer

Then, I set seatTally to *seats* and returned 0.

**Test Cases**

|  |  |
| --- | --- |
| Test | Reason |
| assert(hasProperSyntax("CT5D")); | Basic test to see if the function is working. |
| assert(!hasProperSyntax("CH5D")); | Make sure invalid state codes are invalid. |
| assert(!hasProperSyntax("CT5L")); | Make sure parties outside of D, R, or I are invalid. |
| assert(hasProperSyntax("")); | Empty strings must be true. |
| assert(hasProperSyntax("CT")); | Simply a state code technically is proper syntax. |
| assert(!hasProperSyntax("CT5D10")); | Make sure that state forecasts include both a number and a party. |
| assert(!hasProperSyntax(“CT7D,,,,,CA4D”)); | Multiple commas should be invalid. |
| assert(!hasProperSyntax("CA999D")); | Anything greater than 2 digits in a state forecast should be invalid. |
| assert(!hasProperSyntax("CA99DR")); | A state forecast must have a number before its party. |
| assert(tallySeats("", 'I', seats) == 0 && seats == 0); | Should tally up 0 seats because it is proper syntax, but there are no votes for the ‘I’ party. |
| assert(tallySeats("CA95D", 'd', seats) == 0 && seats == 95); | Test if tallySeats is working to set *seats* for at least one state forecast and returning 0. |
| assert(tallySeats("CA95D,CT9R50D", 'd', seats) == 0 && seats == 145); | Test if tallySeats is working for multiple state forecasts. |
| assert(tallySeats("CZ95D,CT9R50D", 'd', seats) == 1 && seats == -999); | Improper state codes should return 1 and not change the value of *seats*. |
| assert(tallySeats("CA95D,CT9R50D", '%', seats) == 2 && seats == -999); | Test tallying seats with a party that is not alphabetical. |
| assert(tallySeats("CA95D,CT9R50D", 'I', seats) == 0 && seats == 0); | Ensure that *seats* is set to 0 if there are no seats for a party in *pollData*. |