Tommy Hollenberg
CS161 Assign4
Design Document
02/11/2017

Understanding the Problem

This C++ program will be a game of Wheel of Fortune, based on the popular television show. In the game, players must decipher a phrase in which all of the letters have been replaced with underscores. The game can be played by one to three players, and may consist of one or more rounds, each with a unique hidden phrase. During each round, players amass points, but only the player that correctly guesses the phrase gets their round score added to their cumulative score that round. The player with the highest cumulative score at the end of all of the rounds is the winner (not the player that guessed the most phrases). Each player has three options during their turn.

1. Spin the Wheel: The spin lands on a number from 0-21. If the wheel lands on 1-20, the player guesses a consonant they suspect may be part of the phrase. Any occurrences of the consonant (no case sensitivity) are revealed and remain so for the rest of the round. Also, the player's round score is increased by the number of occurrences multiplied by the number they spun and their turn continues. If they spin a 0 or 21, or if there are no occurrences of the consonant they guessed, their turn is over. Additionally, if they spin a 0 they lose all of the points they had accumulated so far in the round.
2. Solve the puzzle: The player makes an attempt to guess the hidden phrase in its entirety. If their guess matches the phrase exactly (this check should be case insensitive), they win the round and get to keep the points they earned that round. (No additional points are awarded for solving the phrase.) If their guess is incorrect, their turn is over.
3. Buy a vowel: For 10 round points, the player can guess a vowel they suspect may be part of the phrase. (The player must have at least 10 points to choose this option.) Any occurrences of the vowel (no case sensitivity) are revealed and remain so for the rest of the round. The cost is fixed at 10 points and the player's turn continues, regardless of how many occurrences of the guessed vowel there are.

Prior to playing the game, a game moderator must input the number of players, the number of rounds, and a valid secret phrase for each round. Valid phrases consist of only letters (A-Z and a-z), spaces, and certain characters (.!?,;:-') used in specific ways, as defined below:

.!? – Must be the last character AND must be preceded by a letter.
,;: - Must not be the first character AND not be the last character, AND must be preceded by a letter AND followed by a space.

- - Must not be the first character AND not be the last character, AND must be preceded by a letter AND followed by a letter.
' - Must not be the first character AND be preceded by a letter, OR must not be the last character AND be followed by a letter.

After each guessed letter, the program must print out the number spun on the wheel (if not buying a vowel), the number of occurrences found, the player's updated round score, and the updated board state (with discovered letters shown and yet to be discovered letters replaced by underscores). When the hidden phrase is correctly guessed, the program must print a message of success for the guessing player and the current cumulative standings and scores for all players. After all rounds have been played, the program must print the final scores and standings and declare a winner.

Devising a Plan

A user-defined **Player** object will be used to hold player information such as player number, round score, and total score. A static counter member variable will be used in the default constructor to initialize player number with unique, sequential values starting from one. Round score and total score will be initialized to zero in the default constructor. All member variables will be type int, because all scoring in this game is integer based.

In **main**, integer variables for the number of players and the number of rounds will be created, as well as a Player object pointer and a string pointer that will later point to dynamically allocated arrays sized based on the user input player and round numbers, respectively. The random number generator will be seeded in main as well. Three user-defined functions will be called from main, which will handle game setup, playing the game, and declaring the winner. Finally, memory will be deallocated. The pseudocode for main is shown below:

```
main() {
   Declare variables
   Seed random number generator (call srand())

   Call game_setup(addresses of variables in main to be initialized)
   Call play_game(initialized variables)
   Call declare_winner(variables, which were changed in play_game)

   Deallocate memory
   return 0;
}
```

Inside **game_setup**, the user will be prompted to input the number of players and the number of rounds. These will both be handled by a **get_integer** function that I have already written for one of the previous assignments. A prompt string and a maximum accepted input integer are passed to get_integer and it continues to output the prompt until the user enters an integer from 1 to the maximum accepted

value, which is then returned to the caller. The maximum accepted number of players will be 3, and the maximum accepted number of rounds will be the default value of the parameter, INT_MAX, which will be defined by a preprocessor #define macro and will be equal to 2147483647, the maximum value of a signed 32-bit integer. Then, memory will be dynamically allocated for a Player object array with length equal to the number of players and a string array with length equal to the number of rounds. A for-loop will iterate through the string array and assign each string a secret phrase returned by the **get_phrase** function, which must be passed a prompt string. The get_phrase function continues to prompt the user to input a string using std::getline until the user enters a valid phrase as determined by the **check_phrase_validity** function. The check_phrase_validity function checks each character in the passed string and ensures that they all follow the rules outlined above in the "Understanding the Problem" section. The **is_alphabetic** function is used to check if a character is a lowercase or uppercase letter, and returns true if the passed character is a letter and false otherwise. If every character is valid and there is at least one letter, check_phrase_validity returns true, and returns false otherwise. Finally, after a valid phrase is entered for every round, the game_setup function uses the system("clear") function in the cstdlib library to hide the secret phrases from the players. The pseudocode for game_setup, get_phrase, check_phrase_validity, and is_alphabetic is shown below:

```
void game_setup(number of players, number of rounds, pointer to Player
  pointer, pointer to string pointer) {
    Output "Wheel of Fortune Game Setup"
    Number of players = get_integer("How many players(1-3)? ", 3)
    Number of rounds = get_integer("How many rounds? ")
    Dynamically allocate Player array of [number of players] elements
      to hold players and assign to the pointer to Player pointer
    Dynamically allocate string array of [number of rounds] elements
      for secret phrases and assign to the pointer to string pointer
    For each round
        Secret phrase for the round = get_phrase("Enter round phrase")
    Clear console to hide secret phrases
}

string get_phrase(prompt string) {
    Declare string to hold secret phrase
    Do
        Output prompt string
        Get user input secret phrase with std::getline()
    Loop while check_phrase_validity(secret phrase) returns false
    Return secret phrase
}
```

```
bool check_phrase_validity(secret phrase) {
    Declare Boolean indicating whether there is at least one letter
      and initialize it to false
    For each character of the secret phrase
        If the character is_alphabetic or a space
            If the character isn't a space
                Letter = true
            Continue to next character
        If this isn't the first character and the previous character
          is_alphabetic
            If this is the last character and it is a period,
              exclamation point, or a question mark
                Continue to next character
            If this isn't the last character, the next character is a
              space, and it is a comma, semicolon, or colon
                Continue to next character
            If this isn't the last character,  the next character
              is_alphabetic, and it is a hyphen
                Continue to next character
        If it is an apostrophe and 1) it isn't the first character and
          the previous character is_alphabetic, or 2) it isn't the
          last character and the next character is_alphabetic
            Continue to next character
        Return false
    Return letter
}

bool is_alphabetic(character) {
    Return the Boolean result of: the character is >= 'A' and <= 'Z',
      or the character is >= 'a' and <= 'z'
}
```

In the **play_game** function, integers to hold the number of turns that have been taken and the number of vowels and consonants that have been guessed will be created. The zero-based number of turns will be used to determine which player's turn it is by:

$$Player\ array\ index\ of\ current\ player = turn\ number\ \%\ number\ of\ players.$$

An integer array (the "alphabet") of length LETTERS_IN_THE_ALPHABET (a preprocessor #define macro equal to 26) will also be created to keep track of which letters have been guessed. All elements will be initialized to zero, to represent yet to be guessed letters. As letters are guessed, the element with an index corresponding to each letter's position in the alphabet will be set to one ('a' is index 0, 'b' is index 1, …, 'z' is index 25). Lastly, a Boolean variable will be created to signal when the hidden phrase has been solved and a string variable will be created to hold the current board state (this will be updated after each successfully guessed letter). A for-loop will be used to iterate through the array of hidden phrases for each round. At the beginning of each round, the "solved" Boolean variable

will be set to false, all players' round scores will be reset to zero, the board state string variable will be set to the hidden phrase for the current round, and the history of guessed letters will be reset using the **reset_guess_history** function. The reset_guess_history function will be a void function that will be passed the addresses of the "alphabet" and the integers tracking the number of vowels and consonants guessed, and will set all elements of the integer array and both integer variables to zero. Before beginning the round, the **censure_phrase** function will be called to iterate through the board state string and replace all letters (as determined by the is_alphabetic function) with underscores. Then a while-loop will be used to increment the number of turns variable and call the **take_turn** function, which will be described below, until the "solved" variable (which will be assigned the Boolean return value of the take_turn function) is true. Finally, the player that guessed the phrase correctly will be congratulated and have their round score added to their total score, before the current standings are output to the console by the **print_standings** function. The print_standings function will create an array of pointers to Players objects with length equal to the number of players. A for loop will be used to assign each pointer of this array to one of the game players. Then, the **sort_by_total_score** function will be used to sort the Players being pointed to by total score, in descending order (highest score is pointed to by element 0 of the Player pointer array). This can be done simply with a switch-case statement due to the number of players being capped at 3, and will utilize the swap() function in the algorithm library. Swapping is necessary if the total score of the player pointed to by first element is less than that of the player pointed to by the second element. The cases for one and two players are trivial and need no explanation. For three players, after the first two swaps (if necessary), the player with the lowest score is guaranteed to be pointed to by the third element. Then, the remaining two players can be handled as if there were only two players, since there is no break statement between case 3 and case 2 (see below for pseudocode). After the players have been sorted in descending order by total score, the print_standings function outputs the current standings, and returns the number of the player with the highest total score. If the top two players have the same total score, a zero will be returned (indicating a tie). This return value will not be used by the play_game function, but will be used by other calling functions. The play_game function has no return value because the Player array was passed by address, so the changes made within play_game will also affect the variables in the calling function. The pseudocode for play_game, reset_guess_history, censure_phrase, print_standings, and sort_by_total_score is shown below:

```
void play_game(number of players, number of rounds, player array,
  secret phrase array) {
    Declare and initialize variables
    Output "Let's play, Wheel of Fortune!"
    For each round
        Set solved to false
        Call reset_guess_history("alphabet", number of vowels guessed,
          number of consonants guessed)
        Set board state to the secret phrase of the current round
        Call censure_phrase(board state)
        For each player
            Set round score to 0
        Output the current round number to the console
        While the secret phrase has not been solved
            Increment the number of turns (which is initialized to -1)
            Solved = return value from call to take_turn(current
              player, board state, secret phrase, "alphabet", number
              of vowels guessed, number of consonants guessed)
        Congratulate player that solved secret phrase
        Add their round score to their total score
        Call print_standings(number of players, player array)
}

void reset_guess_history("alphabet", number of consonants guessed,
  number of vowels guessed) {
    For each element of "alphabet"
      Set element to 0
    Number of consonants guessed is 0
    Number of vowels guessed is 0
}

int censure_phrase(board state: is the secret phrase before censure) {
    Declare integer to hold the number of letters and initialize to 0
    For each character in the board state
        If the character is_alphabetic
            Set the character to an underscore
            Increment the number of letters by one
    Return the number of letters
}
```

```
int print_standings(number of players, player array, final score?) {
    Declare an array of Player pointers ("standings") of length
      [number of players]
    For each element of "standings"
        Point element to the next player in the player array
    Call sort_by_total_score(number of players, "standings")
    Output "Standings:"
    For each player pointed to in "standings"
        Output player number and total score
    If the highest total score is equal to the second highest score
        Return 0
    Otherwise return the player number of the highest scoring player
}

void sort_by_total_score(number of players, standings)
    switch(number of players) {
        case 3: swap players in elements 0 and 1 if necessary
                swap players in elements 1 and 2 if necessary
        case 2: swap players in elements 0 and 1 if necessary
        case 1: break from switch (not necessary to include this
          line)
}
```

In the take_turn function, two Boolean variables indicating whether the secret phrase has been solved and whether the players turn has ended will be created and initialized to false. An integer variable will be created to hold the player's choice. The number of the current player will be output to the console once to notify the player that it is their turn. Then a while-loop will loop until the "end turn" variable is true. Inside the while-loop, the current board state will be output, and the player will be prompted to enter a number from 1-3 using the get_integer function. If they choose 1, the **spin_wheel** function will be called, the return value of which will be assigned to "end turn." If the player chose 2, the **solve_puzzle** function will be called, the return value of which will be assigned to the "solved" variable, and the "end turn" variable will be set to true, because the player's turn is over regardless of whether they correctly guessed the secret phrase or not. If the player did not choose 1 or 2, they chose 3, and the **buy_vowel** function will be called. The buy_vowel function has no return type, and the "end turn" variable will not be altered, because buying a vowel never ends a player's turn. Then, no matter what option the player chose, the **case_insensitive_compare** function will be called to check whether the board state and the secret phrase strings are the same (the case insensitivity of this function will not be utilized here). If they are, "solved" and "end turn" will both be set to true and the current board state will be output, because even though the player may not have chosen to solve the puzzle, their guessed letter completely filled in all remaining unknown characters, so the round, and consequently their turn, are over. Finally, the "solved" variable is returned. The pseudocode for take_turn and case_insensitive_compare is shown below:

```
bool take_turn(current player, board state, secret phrase, "alphabet",
  number of vowels guessed, number of consonants guessed) {
    Declare and initialize variables
    Output player number
    While the player's turn is not over (end_turn is false)
        Output board state
        Choice = return value from call to get_integer("Spin the
          wheel(1), solve the puzzle(2), or buy a vowel(3)? ", 3)
        If choice is 1
            End_turn = spin_wheel(current player, board state,
              secret phrase, "alphabet", number of consonants guessed)
        Else if choice is 2
            Solved = solve_puzzle(secret phrase)
            End_turn = true
        Else (choice is 3)
            buy_vowel(current player, board state, secret phrase,
              "alphabet", number of vowels guessed)
        If case_insensitive_compare(secret phrase, board state)
          returns true
            End_turn = true
            Solved = true
            Output board state
    Return solved
}

bool case_insensitive_compare(s1 and s2 of equal length) {
    For each character i of s1
        If s1[i] doesn't equal s2[i]
            If s1[i] and s2[i] both return true with is_alphabetic()
                If s2[i] is equal to s1[i] - 32 or s1[i] + 32
                    Continue to next i
            Return false
    Return true
}
```

In the spin_wheel function, the number of consonants guessed will be compared to LETTERS_IN_ALPHABET - NUM_VOWELS (NUM_VOWELS will be a preprocessor #define macro equal to 5). If the number of consonants guessed is greater than or equal to the difference of these two macro values, a message stating that all of the consonants have already been guessed will be output and the function will return false (which means the player's turn is not ended). Otherwise, an integer variable to hold the player's random spin will be created. The address of this variable and that of the current player's Player object will be passed to the **random_spin** function. The random_spin function uses rand() % 22 to assign the spin variable a value from 0-21, and outputs the spun value. If spin is zero, the player's round score is reset to 0 and a message informing the player is output. If the spin is zero or 21, the player loses their turn, so and a message informing the player is output and random_spin returns true. If a number from 1-20 is spun, the function simply returns false. Back in the spin_wheel function, if the random_spin function returns true, spin_wheel returns true as

well, because the player's turn is over. Otherwise, the number of consonants guessed is incremented by one and a character variable is declared and assigned the return value of a call to the **get_letter** function with the "vowel flag" set to false. The get_letter function is passed the "alphabet" and a "vowel flag" Boolean variable indicating whether the input character should be a vowel or a consonant. Inside get_letter, a while-loop with no exit condition ("while(1)") will be used to ensure that the user enters valid input. The user will be prompted to enter a vowel or a consonant (depending on the "vowel flag"). If the character is an uppercase letter, its value will be increased by 32, which converts it into the corresponding lowercase letter. The letter's position in the alphabet will be determined by subtracting the char literal 'a' from it and storing the difference in an integer variable. The element in "alphabet" corresponding to this position will be checked, and if the value is non-zero, a message will be output stating the letter has already been chosen before looping back. If the element in "alphabet" is zero, the letter has not been guessed yet this round. Then the return value of the **is_vowel** function will be compared with the "vowel flag". If they are the same, the element corresponding to the guessed letter in the "alphabet" array will be incremented to one, and the letter will be returned to spin_wheel. Once a previously unguessed consonant has been input by the player, an integer variable will be created to hold the number of occurrences of the guessed letter in the secret phrase. This variable will then be assigned the return value of a call to the **decode_phrase** function. The decode phrase function will be passed the board state, the secret phrase, and the guessed letter. It will loop through the secret phrase, looking for instances of the guessed letter or the guessed letter - 32 (the uppercase version). For each instance found, a counter variable will be incremented, and the character at the same index in the board state string will be set to the value of the character in the secret phrase string (whether it be the uppercase or lowercase version of the guessed letter). The decode_phrase function will then return the number of instances found. Back in the spin_wheel function, the player's round score will be increased by the number of occurrences of the guessed letter multiplied by the value they spun on the wheel, and the result of their guess will be output by calling the **print_guess_result** function. This function simply prints the number of occurrences of the guessed letter and the updated round score of the player. Finally, if there were zero occurrences of the guessed letter in the secret phrase, the player is informed that their turn is over and the spin_wheel function returns true. If there were one of more instances of the guessed letter, spin_wheel returns false. The pseudocode for spin_wheel, random_spin, decode_phrase, get_letter, and is_vowel is shown below:

```
bool spin_wheel(current player, board state, secret phrase,
  "alphabet", number of consonants guessed) {
    If the number of consonants guessed is >= LETTERS_IN_ALPHABET -
      NUM_VOWELS
        Output "all consonants have already been guessed"
        Return false
    Declare integer to hold the spin value
    If call to random_spin(current player, spin) returns true
        Return true
    Increment number of consonants guessed
    Declare character to hold guessed consonant
    Consonant = return from call to get_letter("alphabet", false)
    Declare integer to hold number of occurrences
    Num_in_phrase = return from call to decode_phrase(board state,
      secret phrase, consonant)
    Add product of num_in_phrase multiplied by spin to the player's
      round score
    Call print_guess_result(current player, num_in_phrase, consonant)
    If num_in_phrase is zero
        Output "your turn is over"
        Return true
    Return false
}

bool random_spin(current player, spin) {
    Assign spin the result of rand() % 22
    Output "You spun a " and [value of spin]
    Switch(spin) {
        Case 0:  Set player's round score to 0
                 Output "You lose all of your points"
        Case 21: Output "Your turn is over"
                 Return true
    }
    Return false
}

int decode_phrase(board state, secret phrase, letter) {
    Declare integer variable to count number of instances found and
      initialize to 0
    For each character in the secret phrase
        If letter or letter - 32 (the capital version) are equal to
          the secret phrase character
            Set corresponding board state character to the secret
              phrase character
            Increment instances found by one
    Return instances found
}
```

```
char get_letter("alphabet", "vowel flag") {
    Declare integer to hold the letter's position in the alphabet
    Declare character to hold the user input letter
    While (1)
        If vowel_flag is true output "pick a vowel", if false output
          "pick a consonant"
        Get user input letter
        If letter - 'A' is < LETTERS_IN_ALPHABET
            Add 32 to letter (to convert from uppercase to lowercase)
        Position = letter - 'a'
        If position is >= 0 and < LETTERS_IN_ALPHABET
            If is_vowel(position) is equal to "vowel flag"
                If the element of "alphabet" at index position is 0
                    Increment the element of "alphabet" at index
                      position by one
                    Return letter
                Otherwise output "that letter has already been chosen"
}

bool is_vowel(position) {
    Declare an integer array of length 5 and initialize to {0, 4, 8,
      14, 20}, which are the indices of the vowels in "alphabet"
    For each element in the "vowels" (the array declared in the
      preceding line)
        If position is equal to the element in "vowels"
            Return true
    Return false
}
```

In the solve_puzzle function, a string variable is declared to hold the user's solution attempt, and a Boolean is declared to indicate whether the guess was correct or not. The user is prompted to enter their guess, and input is handled with the std::getline() function. The "correct" variable is assigned the return value of case_insensitive_compare passed the user's solution attempt string and the secret phrase string (this usage is why we need a user-defined function and cannot simply use the string::compare() function). An appropriate statement based on the value of "correct" is output to inform the user of their success/failure, and "correct" is returned. The solve_puzzle pseudocode is shown below:

```
bool solve_puzzle(secret phrase) {
    Declare variables
    Output "Enter the phrase: "
    Get user input phrase
    Correct = case_insensitive_compare(secret phrase, user's guess)
    If correct is true
        Output "Correct!"
    Else
        Output "That is incorrect."
    Return correct
}
```

In the buy_vowel function, if the player's round score is less than 10, a message is output stating the player doesn't have enough points to buy a vowel and the function returns to the caller. If they have 10 or more round points, then if the number of vowels guessed is greater than or equal to NUM_VOWELS, a message stating all of the vowels have already been bought is output and the function returns to the caller. Otherwise, 10 points are subtracted from the player's round score and the number of vowels guessed is incremented by one. A character variable is declared to hold the user's guessed vowel, and is assigned the return value of a call to get_letter with "vowel flag" set to true. An integer is declared to hold the number of occurrences of the guessed vowel in the secret phrase, and is assigned the return value of a call to decode_phrase. Finally, the result of the user's guess is output by calling the print_guess_result function. The buy_vowel pseudocode is shown below:

```
void buy_vowel(current player, board state, secret phrase, "alphabet",
  number of vowels guessed) {
    If player's round score is < 10
        Output "You don't have enough points to buy a vowel"
        Return
    If number of vowels guessed is >= NUM_VOWELS
        Output "all vowels have already been bought"
        Return
    Subtract 10 from the player's round score
    Increment the number of vowels guessed by one
    Declare character to hold guessed vowel
    Vowel = get_letter("alphabet", true)
    Declare integer to hold number of occurrences in secret phrase
    Num_in_phrase = decode_phrase(board state, secret phrase, vowel)
    Call print_guess_result(current player, num_in_phrase, vowel)
}
```

Back in the main function, after the play_game function returns, the **declare_winner** function is called. In declare_winner, an integer variable to hold the player number of the winner is declared and assigned the return value of a call to print_standings. If the winner is 0, a message is output stating that the game has ended in a tie. Otherwise, the player with the highest score is declared the winner. The declare_winner pseudocode is shown below:

```
void declare_winner(number of players, player array) {
    Declare integer to hold the player number of the winner
    Assign winner the return value of print_standings(number of
      players, player array)
    If winner is 0
        Output "It was a tie"
    Otherwise output "The winner is player " [winner]
}
```

Testing Plan

Testing get_integer function (assuming max input value is 3):

Input Value:	Expected Outcome:
3	Returns 3 to the caller
1	Returns 1 to the caller
0	Outputs prompt again and waits for further user input
4	Outputs prompt again and waits for further user input
a	Outputs prompt again and waits for further user input
2sd	Returns 2 to the caller and ignores rest of input
5sf	Ignores rest of input, outputs prompt again and waits for further user input
/	Outputs prompt again and waits for further user input

Testing get_phrase function (and consequently testing check_phrase_validity and is_alphabetic as well):

Input Value:	Expected Outcome:
You're the best.	Valid input (returns the phrase)
Catdog	Valid input
Leave. NOW!	Invalid input (outputs prompt again and waits for user input), periods can only be the last char
.	Invalid input, must contain at least one letter
Welcome, ma'am!	Valid input, returns the phrase
47	Invalid input, no numbers allowed
Have some self-respect, man.	Valid input
Yes,no	Invalid input, commas must be followed by a space
Grandma?	Valid input

Testing get_letter function (and consequently is_vowel as well):

Input Value:	Expected Outcome:
vowel_flag is false, previously guessed a, b, c, d, e	
i	Re-output "Pick a consonant: " and wait for input
4	Re-output "Pick a consonant: " and wait for input
a	Re-output "Pick a consonant: " and wait for input
b	Output "That letter has already been chosen.", then re-output "Pick a consonant: " and wait for input
z	Return 'z' to the caller
Z	Return 'z' to the caller
vowel_flag is true, previously guessed a, b, c, d, e	
f	Re-output "Pick a vowel: " and wait for input
/	Re-output "Pick a vowel: " and wait for input
b	Re-output "Pick a vowel: " and wait for input
a	Output "That letter has already been chosen.", then re-output "Pick a vowel: " and wait for input
i	Return 'i' to the caller
I	Return 'i' to the caller

Testing program logic and helper functions:

Situation:	Input Value(s):	Expected Outcome:
"Do you want to spin the wheel(1), solve the puzzle(2), or buy a vowel(3)?"	1	Output result of random spin. Then ask for consonant or output "Your turn is over." and move to the next player
	2	"Guess the phrase: " and wait for input
	3	Check whether the player has enough points, if so, then ask for a vowel
	0	Re-output prompt and wait for input
	F	Re-output prompt and wait for input
	-1	Re-output prompt and wait for input

Testing censure_phrase function

Secret phrase	**Censured board state**
You're the best.	___;__ ___ ____.
Welcome, ma'am!	_____, __'__!
Have some self-respect, man.	____ ____ ____-_____, ___.

Testing decode_phrase function

Secret phrase	**Board state**	**Guessed Letter**	**New board state**	**Instances found**
Lollipop	_o____o_	l	Loll__o_	3
		b	_o____o_	0
		i	_o__i_o_	1
		e	_o___o	0

Testing case_insensitive_compare

Secret phrase	**Guess**	**Result**
Mr. Rogers	mr. rogers	"Correct!"
	Mr Rogers	"That is incorrect."
	mR. rOGERS	"Correct!"
	Mr. R0gers	"That is incorrect."

Testing declare_winner/print_standings/sort_by_total_score

Player	**Total Score**	**Sorted Standings**		**Winner Declaration**
1	10	2	30	"Player 2 is the winner!"
2	30	3	20	
3	20	1	10	
1	30	1	30	"It was a tie!"
2	0	3	30	
3	30	2	0	