What to look for in Problem Set 2: Readability

What to Look For

Evidence of a 5-point Submission

- Efficiently calculates the grade in an easy-to-read manner (equation is not written out on one long line, few or no unnecessary parentheses, no unnecessary numbers or operations included; may include coefficients as magic numbers in this instance, may also explain where the equation came from in a comment)
- Makes use of the **isalpha** function to determine whether a character is alphabetical.
- Counts words without assuming there will always be one space between each word; does not need to start the word counter at one, or add one to it later.

Evidence of a 4-point Submission

- Instead of using **isalpha**, uses the **toupper** or **tolower** function to check values between a single alphabetical range
- Counts words by counting spaces, assuming there will always be one space between each word. Explains why they start the word counter at 1 (or add 1 later) with a comment.
- Uses the **OR** operator to check for multiple values (punctuation, upper and lower case letters) in a single conditional

Evidence of a 3-point Submission

- Calculates sentence, word, and character counts with a **for** loop over the input string; counts sentences and characters using ASCII values explicit character comparison
- Uses unnecessary variables, as by storing array subsets, such as **text[i]**, in a separate variable before passing that variable as input to a function
- Counts words by counting spaces, assuming there will always be one space between each word. Does not explain why they add 1 to the word counter with a comment.

Evidence of a 2-point Submission

- May check for more conditions than necessary when printing the grade
- May use extra, unnecessary variables
- May calculate grade using one long, hard to read line
- Uses an unnecessary amount of conditionals to check for any of the counts
- Runs counting functions, such as **count_words**, more than once.

Evidence of a 1-point Submission

- Checks for conditionals that are logically unsound or confusing
- Grade calculation is long and/or confusing to read
- Checks for a non-alphanumeric character which would incorrectly indicate a sentence

Example Implementations (Worse vs. Better)

count_letters Function

Worse Implementation

The below example uses magic numbers, as well as an unnecessary number of conditional statements. Could also be improved with the ++ operator.

```
...
    if (s[i] >= 61 && s[i] <= 122)
    {
        count = count + 1;
    }
    else if (s[i] >= 41 && s[i] <= 90)
    {
        count = count + 1;
    }
...
```

Better Implementation

The below example uses characters instead of ASCII values as well as the ++ operator, and combines both ranges in one conditional using the **OR** operator.

```
...
    if (s[i] >= 'a' && s[i] <= 'z' || s[i] >= 'A' && s[i] <= 'Z')
    {
        count++;
    }
...
```

The below example only has to check one range, but still requires two function calls and two comparisons.

```
...
    if (toupper(s[i]) >= 'A' && toupper(s[i]) <= 'Z')
    {
        count++;
    }
...
```

Best Implementation

Note in the example below the evaluation of isalpha() as a boolean, without the need for any comparison within the parenthesis.

```
...
    if (isalpha(s[i]))
    {
        count++;
    }
...
```

count_words Function

Worse Implementation

Uses magic numbers and assumes one space per word. Could also be improved using the ++ operator and a comment explaining why they add one to count at the end of the for loop.

```
int count = 0;
for (int i = 0; i < strlen(s); i++)
{
    if (s[i] == 32)
    {
        count = count + 1;
    }
}
count = count + 1;
return count;
```

Better Implementation

Uses the character instead of the ASCII value, as well as the ++ operator. Leaves a descriptive comment explaining the design of starting the count at 1. Still assumes one space per word.

```
// starting at 1 to account for the last word in the sentence
int count = 1;
for (int i = 0; i < strlen(s); i++)
{
    if (s[i] == ' ')
    {
        count++;
    }
}
count++;
return count;
```

Best Implementation

An example of a robust method: does not assume one space per word and does not initialize counting at one.

```
int count = 0;
bool new_word = true;
for (int i = 0, len = strlen(s); i < len; i++)
{
    // Mark next non-space character as a new word
    if (s[i] == ' ')
    {
        new_word = true;
        continue;
    }
    if (new_word)
    {
        count++;
        new_word = false;
    }
}
return count;
```

Grade Calculator

Worse Implementation

The example below is not as easy to read. Note the extraneous parentheses.

```
int coleman = round((0.0588 * (100.0 * (letters / words))) - (0.296 * (100.0 * (sentences / words)) - 15.8));
```

Better Implementation

This example is easy to read.

```
int coleman = round(0.0588 * (100.0 * letters / words)
                    - 0.296 * (100.0 * sentences / words)
                    - 15.8);
```