Engr 101 Project 5: Spaceport Reviews

Program Design Due: Tuesday, November 16, 2021
Project Due: Tuesday, November 23, 2021

Engineering intersections: Data Science/Industrial & Operations Engineering/Computer Science

Implementing this project provides an opportunity to use C++ file I/O, functions, strings, and vectors.

The program design is worth 10 points, the autograded portion of the final submission is worth 100 points, and the style and comments grade is worth 10 points.

You may work alone or with a partner. Please see the syllabus for partnership rules.

Project Roadmap

This is a big picture view of how to complete this project. Most of the pieces listed here also have a corresponding section later in this document that provides more detail.

1. **Read through the project specification (DO THIS FIRST)**
2. **Make sure you understand the NLP keyword search process**
   If you do not understand them, come to office hours so we can help make this part clear.
3. **Complete the program design pseudocode and submit to Gradescope**
4. **Implement Task 1: the `reviews.cpp` helper functions**
   Complete the four required functions that are in the `reviews.cpp` file.
5. **Write Unit Tests for Helper Functions**
   Make sure your helper functions are working properly. See the section on Writing Unit Tests.
6. **Implement Task 2: Read and evaluate the hotel reviews**
   Complete `evaluateReviews.cpp`. Check that you can recreate the `sample_report.txt` file.
7. **Check that your program is sufficiently and correctly commented -- including indenting!**
8. **Submit `reviews.h`, `reviews.cpp`, and `evaluateReviews.cpp` to the Autograder**

Purpose

The purpose of this project is to give you a chance to practice file input and output (file I/O) with file streams and to practice writing your own functions. It also gives more practice using strings and vectors.

Background

Project 4's spaceport has been up and running for 6 months. The company that owns the spaceport wants to analyze the online reviews of the spaceport so that they can improve their customers' experiences. However, they suspect that only some of the reviews are from actual customers, and that the others are fake! Unfortunately, it is difficult for humans to distinguish between truthful and deceptive reviews -- studies show success rates of approximately 50-60%, which is not much better than guessing. It would also be very time consuming to examine each review by hand.

Instead, the company plans to use an automated text-classification algorithm based on machine learning and natural language processing techniques. A challenge with any machine learning approach is finding high-quality training data. Fortunately, a dataset of carefully verified truthful and deceptive reviews is available, thanks to an ancient study from 2011 EY (Earth Years) that investigated reviews of Chicago hotels. Based on the data available in this study, data scientists working for the company have developed a set of keywords that indicate either truthfulness or deception.

You are tasked with writing a computer program to evaluate reviews by automatically identifying these keywords. You have accepted this project, but you need a way to test your algorithm against reviews that you know are truthful and reviews that you know are deceptive. Again, the Chicago hotel review dataset is helpful in testing your algorithm. Once you have implemented the review analysis program, you will evaluate its performance on these reviews.[1]

The Earthlings who originally studied the hotel reviews also developed a website, available in an archived form at http://reviewskeptic.com. We suggest you check it out to get a big-picture idea of the tool you will be implementing.

Your Job

In this project, your job is to write a program that evaluates each of the hotel reviews and categorizes them as truthful or deceptive.

[1] The next step would be to then apply the same program to evaluate the spaceport reviews. But that is a hypothetical scenario which we will NOT be doing for this project. You are ONLY looking at hotel reviews.

Engineering Concepts - Natural Language Processing (NLP)

Natural Language Processing (NLP) is a field at the intersection of computer science and linguistics that designs computer programs that can interpret, categorize, and act on human languages. Human languages are incredibly complex and flexible, and designing a computer program that can understand and respond to even a subset of a single language is notoriously difficult. However, researchers have continually pushed forward in this field, and now NLP has wide-ranging applications including translation between different human languages, virtual assistants, and automated feedback on writing assignments in large courses.

Keyword Search

One of the most basic forms of NLP is to search through a body of text looking for *keywords*, and condition a program's behavior on the presence or frequency of the keywords. Keywords may also have a numeric *weight*, which determines how significant they are. For example, say you are interested in determining the type of pet a person has, based on a written description. So, you ask a bunch of people to describe their pets. Some people tell you what kind of pet they have (e.g. duck, rabbit, horse, cat, dog, snake, etc.), but some don't. Some of them give you sort of a "review" of their pet, and so you need to figure out what kind of pet they have based on a set of keywords.

Let's say you want to determine whether someone has a duck for a pet. You could analyze keywords of pet descriptions that you have verified are about ducks and not-ducks, and then assign a *weight* to each keyword based on how strongly it is associated with ducks as pets (Table 1). Positive values are associated with ducks; negative values are associated with not-ducks. The larger the values (regardless of positive/negative), the stronger the association.

You could then read in another pet's description and find occurrences of the keywords in the list. If the keyword occurs, add its weight to a total running "score" for the pet description. If the total score is positive, it is categorized as a duck. If the total score is negative, it is categorized as a not-duck. See Table 2 for a few examples of this.

Table 1. Example of keywords and weights for describing ducks as pets.

Keyword	Weight	Keyword	Weight
beak	1.2	large	-2.5
egg	1.5	legs	0.5
eyes	0.3	loud	-0.1
feather	0.7	small	0.6
fins	-2.5	tail	0.4
fluffy	-0.3	teeth	-0.5
fur	-1.0	water	1.4
gills	-3.2	wings	4.2

Table 2. Example analysis of pet descriptions using the set of keywords and weights for describing a duck. (In case you're wondering, Pet #3 is a Great Malay Argus.)

Pet #1		**Pet #2**		**Pet #3**	
Description I have to make sure to clean her tank every day, so the water doesn't get dirty. I love to watch her gills move as she breathes and her tail and fins move as she swims. Sometimes, though, I wish she had legs so I could take her for a walk.		**Description** There are fluffy feathers everywhere all the time! He chipped his beak on a large rock the other day. I think I startled him, because his wings flew wide, and then he lost his balance and that's when he chipped his beak.		**Description** This bird is insane. Why did no one tell me it would be so LOUD?? Also, it grew enormous. There should be a warning when you buy it that this is a *large* bird. Not some cute small fluffy thing you can hold on your finger.	
Keywords Found	**Weights**	**Keywords Found**	**Weights**	**Keywords Found**	**Weights**
water	1.4	fluffy	-0.3	loud	-0.1
gills	-3.2	feathers	0.7	large	-2.5
tail	0.4	beak	1.2	small	0.6
fins	-2.5	large	-2.5	fluffy	-0.3
legs	0.5	wings	4.2		
		beak	1.2		
Total Score	**-3.4**	**Total Score**	**4.5**	**Total Score**	**-2.3**
Categorization	**Not Duck**	**Categorization**	**Duck**	**Categorization**	**Not Duck**

As shown here, the choice of keywords and weights is crucial. Therefore, you *must* make sure that your keywords and weights come from a verified set of data and that they have been analyzed properly.[2]

Note: An NLP program that categorizes things based on a keyword search is only as good as its ability to correctly predict things based on those keywords. The percentage of the predictions that are correct is one simple measure of how well an NLP program is working. A perfect NLP program would get 100% of its predictions correct. An NLP program that is no better than random chance would get 50% of its predictions correct.

[2] Rozado, David. "Wide Range Screening of Algorithmic Bias in Word Embedding Models Using Large Sentiment Lexicons Reveals Underreported Bias Types." PloS One, vol. 15, no. 4, PUBLIC LIBRARY SCIENCE, 21/4/2020, p. e0231189, doi:10.1371/journal.pone.0231189. UMLibrary link.

Program Tasks

For this project, you must determine which hotel reviews are truthful and which are not-truthful (deceptive). Do this by applying the NLP keyword search process described earlier. You will be given a set of keywords and weights in a text file named `keywordWeights.txt`. The hotel reviews are in separate files named `review00.txt`, `review01.txt`, etc. Evaluate each available hotel review and write a summary report to `report.txt`. See the following sections for details.

> ***Starter Code***
> Download `reviews.h`, `reviews.cpp`, and `evaluateReviews.cpp` from the Project 5 folder on the Google drive before you read through the program tasks described below. Refer back to these files as you read through Task 1: Reviews Library.

Task 1: Reviews Library (`reviews.h` and `reviews.cpp`)

The first task in this project is to implement a library of several helper functions that support working with and processing the hotel reviews. The **interface** for the library, which consists of function declarations/prototypes, is specified in `reviews.h`. You can refer to this file for an overview of the functions in the library, but **DO NOT change anything in `reviews.h`**.

The actual implementations of the functions for the library are found in `reviews.cpp`. **Two of the functions have already been written for you.** You may call them anywhere in your code as needed. A brief description is here, and more details can be found in the comments in the code:

- `preprocessReview`
  This function modifies a review, represented as a vector of individual words, by changing each word to lowercase letters, removing punctuation, and replacing any strings representing numbers (e.g. "1", "7", "100") with the string "`<number>`".

- `makeReviewFilename`
  This function returns the appropriate file name for a particular review number. For example, `makeReviewFilename(0)` returns "`review00.txt`" and `makeReviewFilename(5)` returns "`review05.txt`".

You are responsible for writing the implementations of the remaining functions in `reviews.cpp`. There are four such functions, listed briefly here and described in more detail in the following sections:

- `readKeywordWeights` - Reads keywords and their weights from an input stream.
- `readReview` - Reads a review from an input stream into a vector of words.
- `wordWeight` - Finds the weight of a given word based on the keywords and their weights.
- `reviewScore` - Computes the score for a review by adding up the weights of its words.

Additional documentation for the functions is also given in the comments in `reviews.cpp`. Read these comments and consider them part of the project specification. You will also conduct unit tests on the individual helper functions so you can ensure they are working correctly.

Read in the Keywords and Their Weights (`readKeywordWeights`)

This is a function that will read in the keywords and their numerical weights from a text file and store each word in the text file to a vector of `string` variables and store the corresponding weights into a vector of `double` variables -- thereby creating 2 parallel vectors for the keywords and their weights. As described in the Required Helper Functions Section, this function is passed a filestream connected to a text file, an empty vector of `string` variables, and an empty vector of `double` variables. The vector of `strings` gets "filled up" by the words in the text file, and the vector of `doubles` gets "filled up" by the numbers in the text file.

Read in a Review (`readReview`)

This is a function that will read in words from a review text file and store each word in the text file to a vector of `string` variables. As described in the Required Helper Functions Section, this function passes in a filestream connected to a text file, and it passes in an empty vector of `string` variables. The vector of `strings` gets "filled up" by the words in the text file.

Calculate a Word's Weight (`wordWeight`)

This is a function that will read in a word from a review and the vectors of keywords and weights. It will compare the review word to each keyword to see if they match. When a match is found, the function will return the weight of the keyword that matches the review word. If no match is found, the function should return 0.0 for the word's weight.

Calculate the Score of a Review (`reviewScore`)

This function takes three parameters: a vector of `strings` representing a review, a vector of `strings` representing the keywords, and a vector of `doubles` representing the keyword weights. It first makes a copy of the review, and preprocesses it. (*Hint: use the `preprocessReview` helper function already written for you.)* If you were to preprocess the original review, you wouldn't have the original version to potentially work with later on. The function will then iterate through each word of the preprocessed review, get its weight using the `wordWeight` function, and add the word's weight to a running total score. After all words are processed, the function returns the total score. For an example of the process of scoring the review, refer to the keyword search process described earlier. Note that in this case, words which are not identified as keywords will simply have a weight of 0.0 returned from `wordWeight`, so that it is safe to add them in without affecting the overall score.

Task 2: Main Program (`evaluateReviews.cpp`)

The main portion of your program will read in the keywords and weights from a file, read in and evaluate hotel reviews from several different files, and write a summary report to `report.txt`.

Read in Keywords

Open a file input stream for the `keywordWeights.txt` file and read each word into a vector of `strings` and each weight in a parallel vector of `doubles`. If the file cannot be opened, output `"Error: keywordWeights.txt could not be opened."` to `cout` and use `return 1;` to exit the `main` function (recall that a nonzero return value from main reports an error).

Read and Evaluate Review Files

The hotel review files are numbered consecutively (e.g. `review00.txt`, `review01.txt`, `review02.txt`, etc.). For each review, open the file and store each word in a vector of `strings`, calculate the score of the hotel review and then categorize the review as follows:

- **truthful**: score > 3.0
- **deceptive**: score < -3.0
- **uncategorized**: otherwise

Repeat this NLP evaluation process for each review until all reviews have been evaluated. You know that you have processed all the reviews once you try to open a file input stream for the next review file and it does not open successfully (because the file doesn't exist!). You don't need to print an error message in this case - simply have your program stop trying to read more reviews.

You are guaranteed that there will always be at least one review and that there will be no gaps in the numbering of the hotel reviews. There will be no more than 100 hotel reviews.

Generate Summary Report

All output from the `evaluateReviews` program is written into a summary report called `report.txt`. No output needs to be written to `cout` (the *only* exception being the error message printed in the case that the keywords file cannot be opened). This summary report contains information about each review that was analyzed as well as aggregate data (e.g. the total number of truthful vs. deceptive reviews, etc.) See the Output File Section for details.

> ***More Helper Functions***
> You may write additional helper functions in `evaluateReviews.cpp`! (See the helper functions section for some suggestions.) However, do not add additional functions to `reviews.cpp` that you intend to use in `evaluateReviews.cpp`, since this would require modifying the `review.h` file to contain their prototypes as well, which is prohibited for this project.

Program Description

In general, you need to write a C++ program that:

1. Open the keyword weights input file; if the file cannot be opened, gracefully exit the program with an error message and **`return 1;`**.
2. Reads in all of the keywords and weights
3. Reads in a hotel review and evaluates its score based on the keywords and weights
4. Repeat step 3 for all review files
5. Write out a summary of the truthfulness and deceptiveness of the reviews.

Note: The above list can be considered to be the *start* of a program design for this project. **You will be required to submit a full program design as part of this project.** Refer to the Program Design Section for more details on what you need to submit.

Terminology

In this program description, the following terms will be used:

- **Required Functions:** Functions required by the project. These functions must use the interface defined. **These functions must be included in the `reviews.cpp` file.**
- **Function Interface:** The first line of a function that defines the function's return type, name, and parameters.
- **Parameters:** Variables that are passed into your function when it is called. These variables are defined in the function interface and may be assumed to exist when the function is called.
- **Input Files:** Files that will be used by your program, such as a `.txt` file.
- **Output Files:** Files that will be produced by your program, such as a `.txt` file.

Required Program

There are many files that are part of this project, but there are two C++ files that you need to write: `reviews.cpp` and `evaluateReviews.cpp`.

The `reviews.cpp` file contains all of the *required* helper functions used in this project (as described in Task 1). The `reviews.cpp` file will be used to create a library of functions for you to use later in evaluating the reviews. The helper functions that you need to complete are located at the top of the `reviews.cpp` file. The provided helper functions are at the bottom of the file; please feel free to look at these provided functions to see what they do, but do not change them.

The `evaluateReviews.cpp` file contains your `main()` function and is the primary program file. Any helper functions that you write in addition to those required should be correctly declared and defined in the `evaluateReviews.cpp` file.

Compiling and Running the Program

To compile the program, use the following compile command:

```
g++ -std=c++11 -Wall -pedantic evaluateReviews.cpp reviews.cpp -o evaluateReviews
```

Note that **both** `evaluateReviews.cpp` and `reviews.cpp` are included in the compile command, but **not** `reviews.h`. Header files are incorporated using `#include` at the top of `.cpp` files, but are never provided to the compile command directly.

Additionally, since the starter code for the project uses some features only available in C++11 (a more modern version of the language), we need the `--std=C++11` flag.

Once compiled, the program can be run from the command line with:

```
3
```

Writing Unit Tests

When working with complex programs made up of several different functions, it's important to be able to test each function individually to make sure it is working correctly on its own. This is called **unit testing**. A strategy for implementing a set of unit tests is to write a separate `main` function (in a different file) that intentionally calls each function one at a time on a variety of inputs and confirms that the outputs from the functions match the expected correct answer.

A few sample unit tests can be found in `unit_tests.cpp` file provided with the project. The comments included in that file describe the way the unit testing process works. We highly encourage you to use these samples as a starting point and write additional unit tests of your own.

In order to compile and run the unit tests, use the following commands:

```
g++ -std=c++11 -Wall -pedantic unit_tests.cpp reviews.cpp -o unit_tests

./unit_tests
```

Note the difference from the compilation command for the regular program. We've basically kept all the review functions from `reviews.cpp`, but we've swapped in `unit_tests.cpp` for `evaluateReviews.cpp`, which means the `main` function containing the tests will be used instead.

You **do not** need to turn in the `unit_tests.cpp` file to the autograder.

Input Files

There is one input file containing the keywords and their weights: `keywordWeights.txt`. There will also be an unknown number of files that contain the hotel reviews; the hotel reviews all have a similar naming convention of `review00.txt`, `review01.txt`, etc. The next sections give more detail on these input files. The input files are located in the Project 5 Google Drive folder.

Keywords and Weights File

The `keywordWeights.txt` file contains a list of keywords and weights that resulted from analyzing an initial set of verified hotel reviews as training data. Each keyword will be on its own line, and there will be no phrases (e.g. "good" and "view" instead of "good view"). Read in all the keywords and weights in the file, no matter how many keywords there are, and store them in parallel vectors of `strings` and `doubles`.

INPUT: `keywordWeights.txt`

```
<keyword 1> <score 1>
<keyword 2> <score 2>
<keyword 3> <score 3>
<keyword 4> <score 4>
<keyword 5> <score 5>
<...>
```

Review Files

Each review file contains the text of a hotel review. The text is in "paragraph" form. Read in each word and store it in a vector of `strings`.

INPUT: `reviewXX.txt`

```
<word 1> <word 2> <word 3> <word 4> <word 5> <word 6> <word 7> <word 8> <word 9> <word 10>
<word 11> <word 12> <word 13> <word 14> <word 15> <word 16> <...>
```

When evaluating the reviews, start with `review00.txt`, read in the words of the review, and evaluate it according to the Evaluate Review algorithm. Then, go to the next review. Keep processing reviews until the next one is not available.

You are guaranteed that there will always be at least one review and that there will be no gaps in the numbering of the hotel reviews. There will be no more than 100 hotel reviews.

You are provided with 20 hotel reviews as test data. Reviews 0-9 are actually truthful; Reviews 10-19 are actually deceptive. (However, the algorithm you implement doesn't categorized them all perfectly correct, as you can see from the sample output provided below.)

Output File

There is one output file for this program: `report.txt`. This file contains a summary report of your hotel review analysis. The summary report should contain:

- A header line, reading exactly `"review score category"`
- A line of information for each report including the following (separated by spaces)
    - Review number (0, 1, 2, etc.)
    - The overall score of the review
    - The categorization (truthful, deceptive, or uncategorized)
- [an extra blank line]
- The total number of reviews analyzed
- The total number of truthful reviews
- The total number of deceptive reviews
- The total number of uncategorized reviews
- [an extra blank line]
- The number of the review with the highest score (You may assume there are no "ties".)
- The number of the review with the lowest score (You may assume there are no "ties".)

SAMPLE OUTPUT: `sample_report.txt` (available on the Google Drive)

```
review score category
0 14.88 truthful
1 3.33 truthful
2 15.68 truthful
3 4.43 truthful
4 4.14 truthful
5 11.29 truthful
6 20.61 truthful
7 -2.89 uncategorized
8 2.71 uncategorized
9 11.93 truthful
10 0.03 uncategorized
11 -13.06 deceptive
12 -3.66 deceptive
13 -8.46 deceptive
14 5.18 truthful
15 -18.68 deceptive
16 -17.88 deceptive
17 -21.61 deceptive
18 -11.25 deceptive
19 -15.08 deceptive

Number of reviews: 20
Number of truthful reviews: 9
Number of deceptive reviews: 8
Number of uncategorized reviews: 3

Review with highest score: 6
Review with lowest score: 17
```

Hints for a Successful Program

Here is some additional information that will help you be successful in this particular project.

Writing to a File in a Loop

This project requires you to write to a file from within a loop. Sometimes, during the course of development, an infinite loop may slip through your careful debugging and cause a file to grow significantly larger than you would want. Here are a few hints to detect if this is happening:

1) Your program takes longer than a second to run. Unlike project 4, this project shouldn't take more than a second (two at the **absolute most**).
    a. If your program takes longer than 2 seconds to run, type CTRL+C to cancel the running program.
2) Your `report.txt` file takes up a lot of memory.

If you are on your own computer, check how big the `report.txt` file is. If it is more than, say, 30 Mb, then you have an infinite loop. Delete `report.txt`.

If you are on a CAEN machine, navigate into the directory where your Project 5 files are located and type the following command into the linux terminal:

```
bash-4.2$ ls -lh
total 1.1G
...
-rwxr-xr-x. 1 your_uniqname users 9.4K Nov 18 18:11 evaluateReviews.cpp
-rw-r--r--. 1 your_uniqname users  223 Nov 15 09:22 readKeywordWeights.cpp
-rw-r--r--. 1 your_uniqname users  356 Nov 16 12:56 readReview.cpp
-rw-r--r--. 1 your_uniqname users 1.0G Nov 18 18:20 report.txt
-rw-r--r--. 1 your_uniqname users  447 Nov 16 15:04 reviewScore.cpp
...
```

The `report.txt` file listed above has a file size of 1.0Gb (yikes!) and should be deleted. To delete the file, use the *rm* command:

```
bash-4.2$ rm report.txt
```

Passing Filestreams to Functions

One of the required functions needs a filestream passed to it, and you may potentially write a helper function or two that also requires a filestream to be passed in. Remember that filestreams are linked to a specific file; therefore you need to pass the filestreams by reference (*not* pass by value) so that the function has access to the file that was already opened by your program.

Undefined Values for Variables

Using a variable that has been declared, but does not yet have a value assigned to it, can cause unexpected behavior from your program. Similarly, if you write a function that has a return variable (e.g. an `int`, `double`, `bool`, etc. function), and you try to return a variable that does not have a value -- or you forget a return statement entirely -- then you can also see unexpected behavior. Some compilers will automatically assign a zero to some types of data, but others will not; therefore, you should always properly assign/initialize values to your variables. If you see that your program outputs big weird numbers on the autograder, it's likely due to an uninitialized variable.

Finding the Minimum/Maximum of a Set of Data

When searching, or keeping track of, the maximum/minimum of a set of data (such as the score of the hotel reviews), you generally are comparing the current value to whatever is the "current highest" or the "current lowest" value. However, for the first value, you don't yet have anything to compare it to. A common best practice is to initialize the "current highest" and/or "current lowest" value to be the first value in your data set. This way, no matter what the actual values in your data set are, you will always be able to find the maximum or minimum value.

Additional Helper Functions

There are four required helper functions for this project that you have to write plus two helper functions that are provided to you; however, you can (and should!) look to abstract other chunks of code into helper functions. Helper functions make your code easier to read and understand and easier to debug. Some ideas for other helper functions are functions that would:

- Categorize a review
- Find the review with the highest score
- Find the review with the lowest score
- etc.

It is up to you how you want to abstract portions of your code; there is no "right" answer and no "wrong" answer. The only reason we're requiring a few specific helper functions for this project is to enforce practice with abstraction and function writing. In general, you can design your functions however you like!

Pass by Value vs. Pass by Reference

When writing your own helper functions, always consider whether to pass parameters *by value* or *by reference* (including `const` reference). If you are unsure, refer back to the Runestone chapter that had the decision tree about pass by value vs. pass by reference.

Submission and Grading

This project has four deliverables: the program design (pseudocode) and the necessary program files: `reviews.h`, `reviews.cpp`, and `evaluateReviews.cpp`.

(You should still submit `reviews.h`, even though you aren't supposed to modify it - the autograder will double check that the file is the same to verify you haven't accidentally changed it, since this could mess up the rest of your code.)

Grades are broken down into three categories: the program design, the autograder, and the style and comments. First, a grader will evaluate the *general completeness and level of detail* of your program design that is submitted to Gradescope and will provide a maximum score of 10 points. Second, the Autograder will be responsible for evaluating your submission and will provide a maximum score of 100 points. Third, a grader will evaluate your submission for style and commenting and will provide a maximum score of 10 points. Thus, the maximum total number of points on this project is 120 points. Each test case on the Autograder is worth the same amount. Some test cases are public (meaning you can see your output compared to the solution as well as the inputs used to test your submission) while many are private (you are simply told pass/fail and the reason). The breakdown of points for each category is described in the next sections.

Program Design/Pseudocode

Maximum Score: 10 points

This project implements a complex algorithm. The program itself is broken up into several functions, so a clear program design is essential to successfully completing this project. For this project, we will be using *pseudocode* for our program design.

Pseudocode is an informal, code-like language used to describe high level details in program design. It does not need to "compile" or conform to strict standards, but it does use the same sort of constructs (e.g. iteration, branching) to convey the control flow of our program. Function calls provide abstractions. This project focuses a lot on the design and layout of the helper functions in addition to the main program. A flowchart that covers all of this would be large and difficult to follow. Pseudocode is a better choice because it can be broken up into different “design chunks” for each of the helper functions. There are guidelines on how to write pseudocode in the Project 5 Google Drive folder.

Starter pseudocode is available in the Project 5 Google Drive folder. We have included part of the pseudocode along with a couple of steps to guide you. **You need to complete the missing parts of the pseudocode.** Be sure to document the steps in the four required functions as well as the main program. If you plan to write any helper functions other than the required helper functions, document those steps as well (you do NOT have to include pseudocode for helper functions that are already written for you). The more detailed your steps are in this pseudocode, the easier it will be for you to implement as actual code. Remember that program designs are personal, living documents -- arrange yours in whichever way makes sense to you (and your partner if you have registered one). When you start to write actual code, you may change your mind about how you want to accomplish something in the program. That's okay! Update your pseudocode (for your own records, you do NOT need to re-submit) and then implement the new approach. If you have questions or want to get feedback on your pseudocode, come to office hours!

Submitting your assignment to Gradescope

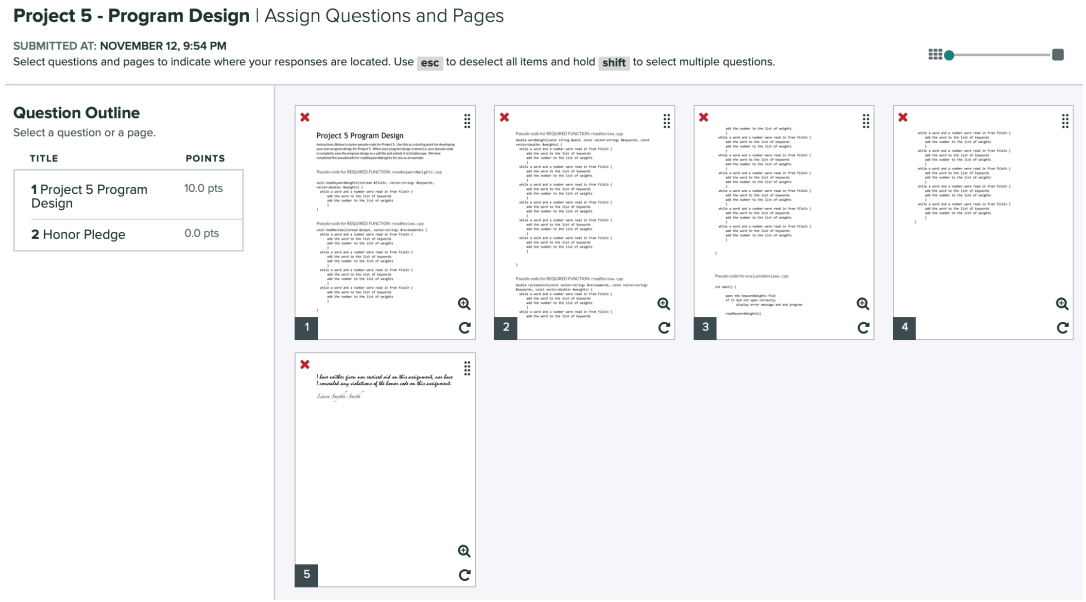
This Gradescope assignment has two questions: the pseudocode and the Honor Pledge. Create a separate document that states the Honor Pledge and includes your signature below the Honor Pledge (if your signature does not look like your name, also print your name).

> If working alone:
>
> > *I have neither given nor received unauthorized aid on this assignment, nor have I concealed any violations of the Honor Code on this assignment.*
>
> If working with a partner:
>
> > *We have neither given nor received unauthorized aid on this assignment, nor have we concealed any violations of the Honor Code on this assignment.*

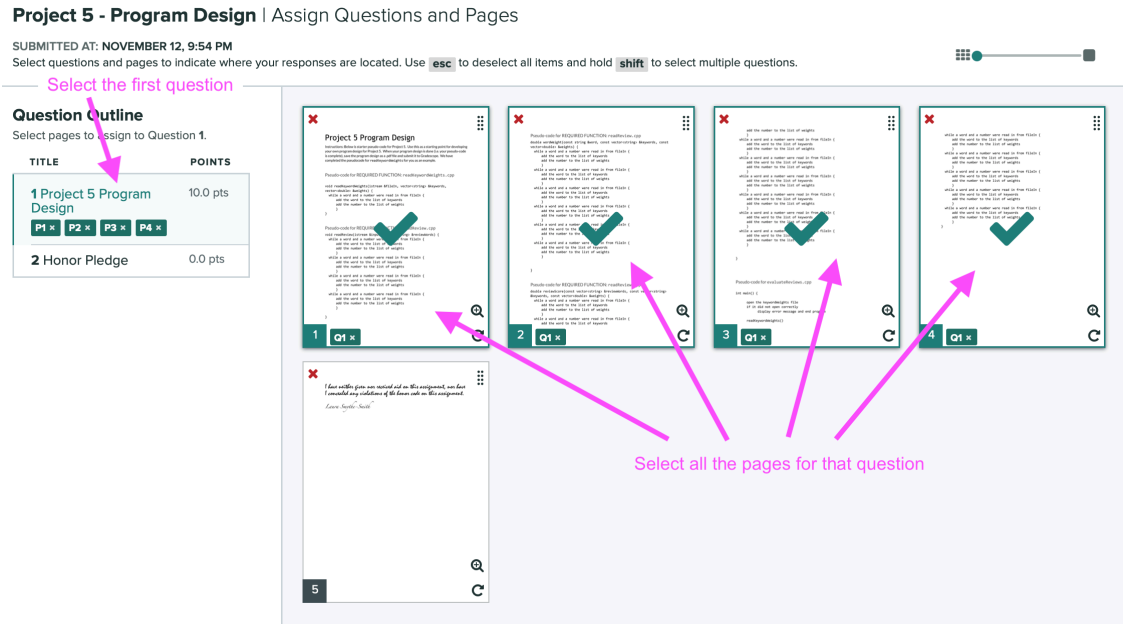
The Honor Pledge and your signature must be *written*, not typed; if you are working in a partnership, both partners must sign the Honor Pledge. To create this written document, you have two options. Option 1: write the Honor Pledge and sign it on a piece of paper and then scan it to create an electronic copy; we recommend the Scannable app to get a high quality scan (low quality scans will not be accepted). Option 2: on a tablet or touchscreen, write the Honor Pledge, sign it, and save the file electronically. Please come to Office Hours if you need help with either of these options.

Create a document that contains both the Project 5 pseudocode and a signed copy of the Honor Pledge. Save this document as a single `.pdf` (the name of this file does not matter).

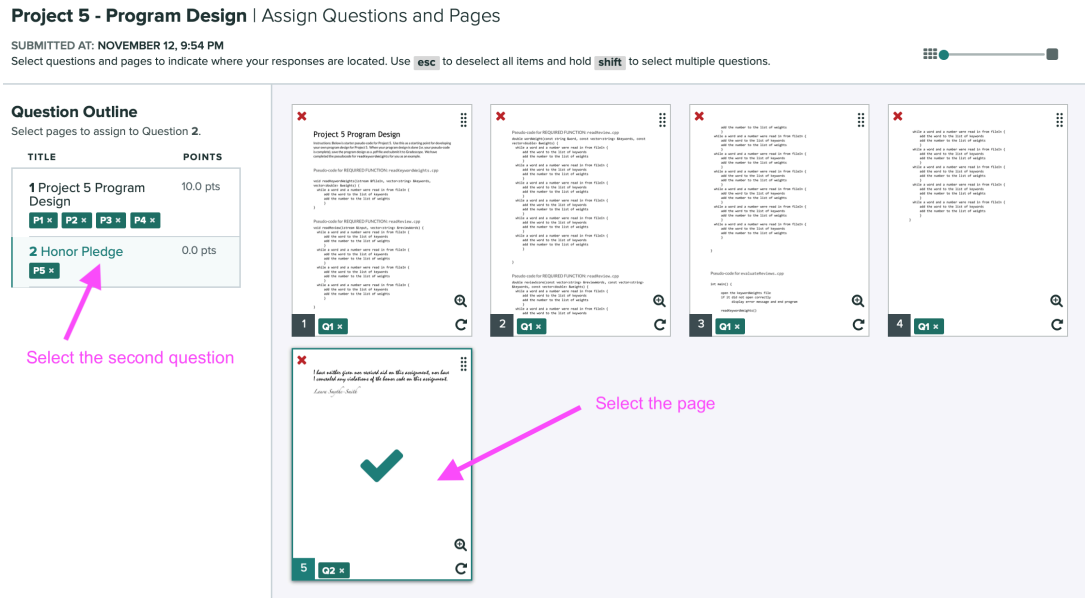
When you upload your assignment, you will see something similar to what is shown below:



This is a "variable length" assignment. Your pseudocode can be as many pages as you need it to be: there is no minimum and no maximum on length. If you are looking for guidance, we would expect the pseudocode portion of the assignment to be at least 2 pages (that's how long the starter pseudocode is!) and probably not more than 5 or 6 pages, but fewer pages is perfectly fine as long as your pseudocode sufficiently details your algorithms and program design. However many pages your pseudocode is, assign all of those pages to the Project 5 Program Design question, as shown below:



Then, assign the page that has your signed honor pledge to the Honor Pledge question, as shown below:

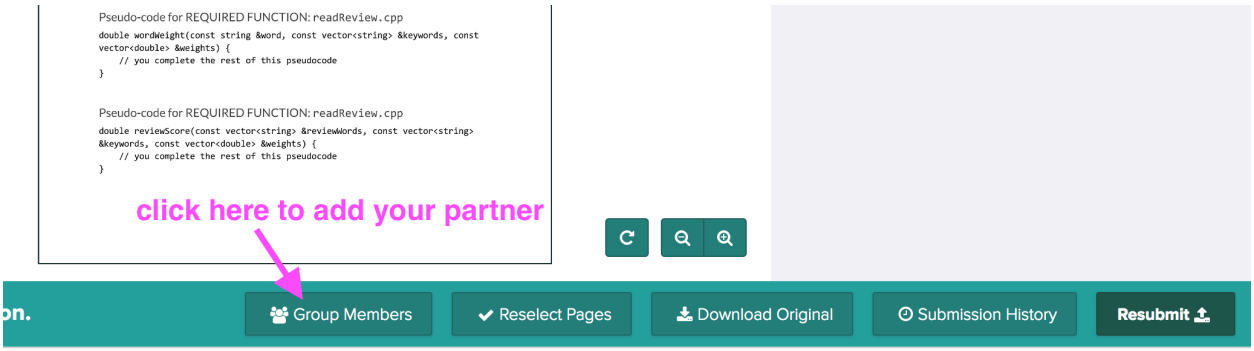


After you have assigned the correct pages to the correct questions, click the "Submit" button in the bottom right corner of the page.

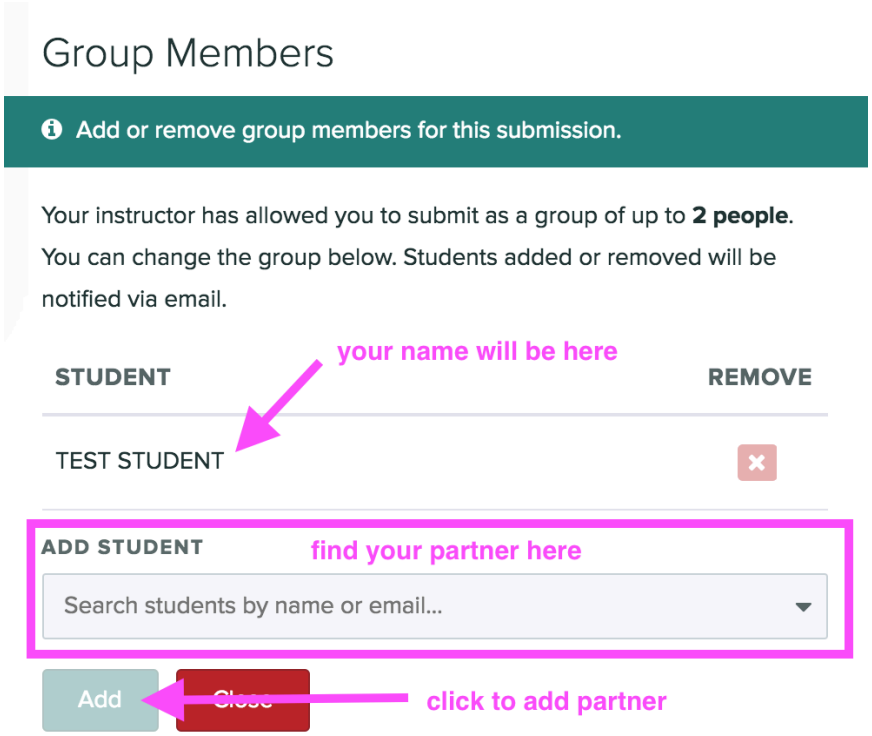
Note: The program design is worth 9 points, and the signed Honor Pledge is worth 1 point (we haven't updated these screenshots yet).

Registering your partnership on Gradescope

If you are working with a partner, use "group submission" on Gradescope. Have one member of the partnership submit the program design and honor pledge as described above. After you submit the `.pdf` file, click "Group Members" to add your partner, as shown below:



This will pop up a little box where you can search for your partner's name and then add them:



Your partner will receive an email verifying that they have been added to your group. Make sure your partner gets this email!

Autograder

Maximum Score: 100 points

Submit your `reviews.h`, `reviews.cpp`, and `evaluateReviews.cpp` files to the Autograder (autograder.io) for grading. The autograder provides you with a final score (out of 100 points) as well as information on *a subset of its test cases*. We provide this autograder to give you a sense of your current progress on the project.

Submitting files to the Autograder

The Autograder will look for three files: `reviews.h`, `reviews.cpp`, and `evaluateReviews.cpp`.

Note: You don't *have* to wait until you have all the files ready before submitting to the Autograder. You can submit a subset of files and get feedback on the test cases related to those files. However, **to receive full credit, you need to submit all files and pass all test cases within a single submission.**

Project scoring and feedback on the Autograder

You are limited to 5 submissions on the Autograder per day. After the 5th submission, you are still able to submit, but all feedback will be hidden other than confirmation that your code was submitted. The autograder will only report a subset of the tests it runs. It is up to you to develop tests to find scenarios where your code might not produce the correct results.

You will receive a score for the autograded portion equal to the score of your best submission. Your latest submission with the best score will be the code that is style graded. For example, let's assume that you turned in the following:

Submission #	1	2	3	4
Score	50	100	100	50

Then your grade for the autograded portion would be 100 points (the best score of all submissions) and Submission #3 would be style graded since it is the latest submission with the best score.

Please refer to the syllabus for more information regarding partner groups and general information about the Autograder.

Style and Commenting

Maximum Score: 10 points

2 pts - Each submitted file has Name, Partner Uniqname (or "none"), Lab Section Number, and Date Submitted included in a comment at the top
2 pts - Comments are used appropriately to describe your code (e.g. major steps are explained)
2 pts - **Indenting** and whitespace are appropriate (including functions are properly formatted)
2 pts - Variables are named descriptively
2 pts - Other factors (variable names aren't all caps, etc...)

Appendix: Test Cases

Test cases are very important for this project. You should first use unit tests for your helper functions so you know they are working correctly. Once you have verified that your helper functions work correctly, continue developing your code and make sure you can recreate the `report.txt` file shown in the test case for evaluating reviews.

Test Case for Evaluating Reviews

You are provided with 20 hotel reviews as test data. Reviews 0-9 are known to be truthful and reviews 10-19 are known to be deceptive; although, your algorithm probably won't be able to correctly categorize ALL of the reviews. We have provided `sample_report.txt` with the starter files, which contains the correct output (also shown earlier in this project spec).