Project 2 – Personality Quiz
CS 251, Fall 2021, Reckinger

Collaboration Policy: By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff. Also, please see copyright at the bottom of this description. Do not post your work for this class publicly, even after the course is over.

Late Policy: You are given a total of 5 late days to use at your discretion (for all projects, not each project). You may use the late days in 24-hour chunks (either 1 day at a time or all five at once). You do not need to alert the instructor to ask permission to use late days. You can manage your use of late days on Mimir directly.

Early Bonus: If you submit a finished project early (by Wednesday, September 15th at 11:59 pm), you can receive 10% extra credit. In order to receive the early bonus: (1) your submission needs to pass 100% of the tests cases; (2) you may not have any submissions after the early bonus deadline.

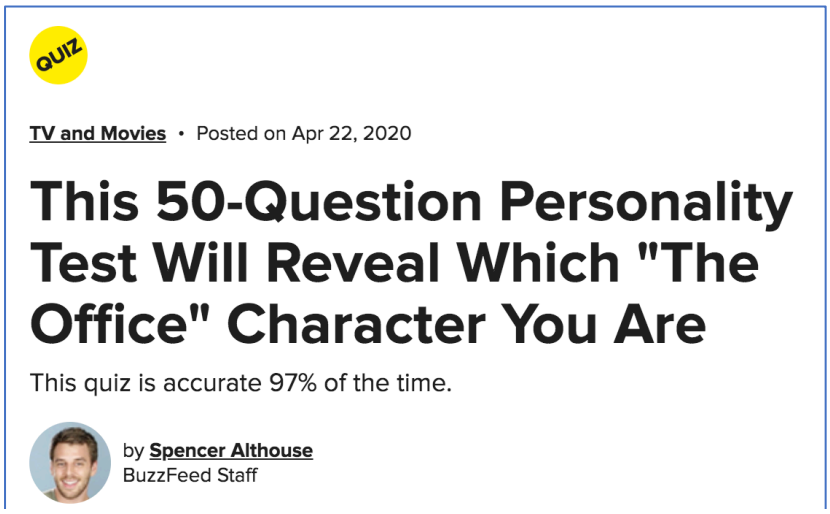
Test cases/Submission Limit: Unlimited submissions. Some test cases hidden.

What to submit: main.cpp, driver.h, mine.people

.pdf starter code

Project Summary

Have you ever been browsing through social media and seen a post like this one?



If you've ever clicked on one of those links, you've probably been presented with a series of statements and asked how much you agree with each of them. At the end, you're given your classification – whether that's which Hogwarts house you're in, which Game of Thrones character you most resemble, or even weirder things like which sandwich best represents you.

Many personality quizzes – including the one you'll be building – are based on a what's called the ***five factor model***. In that model, personalities are described by giving numbers (positive or negative) in five different categories with the handy acronym "OCEAN:" openness, conscientiousness, extraversion, agreeableness, and neuroticism. (It may seem reductionist to distill entire personalities down to numbers in five categories, but hey, the program you're writing is for entertainment purposes only.) As you take the personality quiz and answer questions, the computer updates your scores in these five categories based on how you answer each question. Your score in each category begins at zero and is then adjusted in response to your answers. At the end of the quiz, the program reports the fictional character whose scores are most similar to yours.

Each quiz question is represented as a statement, along with what categories the question asks about and in what way those questions impact those categories. For example, you might have a question like

+A +O I love to reflect on things.

If you got this question on a personality quiz, it might be given to you like this:

How much do you agree with this statement?
I love to reflect on things.
- ○ Strongly disagree
- ○ Disagree
- ○ Neutral
- ○ Agree
- ○ Strongly agree

As indicated by the +A +O here, this particular statement is aligned in the positive direction of categories A (agreeableness) and O (openness), so the way that you answer this question will adjust your scores in categories A and O. For example, suppose you choose "Agree." Since you agreed with the statement, you'd get +1 point in category A and +1 point in category O. Had you chosen "Strongly Agree", you'd instead get +2 points in category A and +2 points in category 0. Choosing "Neutral" here would leave your score unchanged.

On the other hand, if you chose "Disagree," then you'd get -1 point in category A and -1 point in category O. Finally, if you chose "Strongly Disagree," then you'd get -2 points in category A and -2 points in cate - gory O. (Negative numbers don't represent negative personality traits. Positive scores in category O indicate a low barrier to trying new things, while negative scores indicate more caution in trying new things. Neither of these is the "right" way to do things.)

Here's another sample question:

-N +C I try to impress others.

This would be presented to the user in the following way:

How much do you agree with this statement?

I try to impress others.

- ○ Strongly disagree
- ○ Disagree
- ○ Neutral
- ○ Agree
- ○ Strongly agree

The -N +C here means that this question is designed to count negatively in category N (neuroticism) and positively in category C (conscientiousness). So, for example, answering with "Strongly Agree" would add -2 to your N score and +2 to your C score, and answering with "Strongly Disagree" would add +2 to your N score and -2 to your C score. Answering with "Neutral" wouldn't change your score.

As a last example, consider this question:

+E I am the life of the party.

The +E on this question means that it counts positively in factor E ("extraversion"). An answer of "Neutral" wouldn't impact your score in the E category. An answer of "Agree" would add +1 to your E score, and an answer of "Strongly Disagree" would add -2 to your E score.

The Project at a Glance

This project is broken down into milestones. Each milestone should be fully tested by you before moving on to the next milestone. In addition, there are a set of test cases for each milestone provided in the autograder on Mimir.

Your first task is to implement the data processing framework that works behind the scenes to tabulate scores from the personality quiz and determines who the user is most similar to. All of this is contained in driver.h. The blank functions can be found in the starter code. Do not change the function headers (all parameters, function names, returns, etc. must remain as given). Your code should adhere to all efficiency and style guidelines

Your second task is to use your code in driver.h to build the user interface. Therefore, your functions in driver.h will be used to select which questions to ask the user, to determine the user's score from their quiz answers, to determine the similarity between the user and a fictional character, and to select the fictional character who is most similar to the user. You will need to write this user interface from scratch in main.cpp. Your code should be properly decomposed and should adhere to all efficiency and style guidelines.

Milestone 1: Select Random Questions

In order to give a personality quiz, you'll need some way of selecting which questions to ask the user. To do this, we'd like you to write a function

```
Question randomQuestionFrom(set<Question>& unaskedQuestions);
```

that works as follows. We provide as input to this function a set containing objects of type Question, where Question is a custom type we'll detail a bit later in the assignment. This set represents a bank of personality questions that haven't yet been asked. You should implement the function to do the following: choose a random question from the set, remove it from the set, then return it. The idea is that someone could call this function multiple times to pick personality quiz questions to ask the user. You should use the `randomElement` function, which is provided in "driver.h". This function takes in a set and returns (but does not remove) a random element of that set.

***NOTE**: `randomElement` uses a function from myrandom.h. If you take a look at myrandom.h, you will notice there is a Boolean variable called `useAutograder`. If `useAutograder` is true, the randomness is seeded. This means the question set and order will be the same every time the program is run. This is what it will be set to in the autograder (however, we have our own copy, so it doesn't matter what your version is set to). If `useAutograder` is false, then a different set of questions with a different order will be used each time the program is run. That is more fun and probably what you want. But feel free to change it for your own testing purposes.*

If the input set provided to `randomQuestionFrom` is empty, then you should throw an error (check out how it is done in ourvector.h from Project 1) – after all, it's not possible to pick anything out of an empty set.

To recap, here's what you need to do in the first step:

1. Implement the `randomQuestionFrom` function in driver.h.
2. Use the provided tests to confirm that your code works correctly.
3. Test on your own by calling it in main.cpp and make your own tests to make sure it is working.

Milestone 2: Compute Scores From Question/Answer Pairs

In the previous milestone, you worked with variables of type `Question` without looking too closely at how the Question type works. For the next step, you will need to manipulate Questions, so let's dive deeper into this type. We've defined the `Question` type in driver.h, and it looks like this:

```
struct Question {
  string questionText;
  map<char, int> factors;
};
```

Here, `questionText` contains the text of the personality quiz question, and factors is a map encoding the question's assessed OCEAN factors. For example, the question

-E +N I let others finish what they are saying

Would be represented as a `Question` where `questionText` is "I let others finish what they are saying" and where factors is a map that associates 'E' with -1 and and 'N' with +1.

Here's a quick sample of a variable of type Question in use:

```
Question q = /* ... get a question from somewhere ... */
cout << q.questionText << endl;
if (q.factors.count('O') == 1) {
  cout << "O score for this question: " << q.factors['O'] << endl;
}
```

Now, on to what you need to do for this milestone. The code that administers a personality quiz will record the user's answers in a variable of type map. Here, the keys represent individual personality quiz questions, and the values represent the result the user typed in. As a refresher, a response of 1 means "strongly disagree," while a response of 5 means "strongly agree." Your task is to write code that uses the data in that map to determine the user's OCEAN scores. Specifically, we'd like you to write a function

```
map<char, int> scoresFrom(map<Question, int> &answers);
```

that takes as input a map containing the user's answers to their personality quiz questions, then returns a map representing their OCEAN scores. You should compute scores using the approach described earlier: for each question, weight the factors of that question based on the user's answer (5 means "strongly agree" and adds doubles the weight of each factor; 4 means "agree" and adds the weight of each factor; 3 means "neutral" and adds zero to the weight of each factor, etc.).

As an example, suppose you were provided these questions and the indicated responses:

Question	*Factors*					*Answer*
	O	***C***	***E***	***A***	***N***	
I am quick to understand things.	+1				+1	5
I go my own way.	+1		-1			4
I know no limits.			1	-1		2
I become overwhelmed by events.	-1				-1	1

You should then produce a map where O maps to +5, E maps to -2, A maps to +1, and N maps to +4. To see where this comes from, let's focus on the O component. The answer of 5 to "I am quick to understand things" adds 2 × (+1) = +2 to the O score, the answer of 4 to "I go my own way" adds 1 × (+1) = 1 to the O score, and the answer of 1 to "I become overwhelmed by events" adds (-2) × (-1) = +2 to the O score. Running similar calculations for the other factors accounts for the other numbers – do you see why? Also, notice that C wouldn't be a key in the resulting map, since none of the questions have C as a factor.

Similarly, suppose you were given these question/response pairs:

Question	*Factors*					*Answer*
	O	***C***	***E***	***A***	***N***	
I keep my thoughts to myself.			-1			4
I show my gratitude.			1	1		3
I put off unpleasant tasks.		-1				4
I take an interest in other people's lives.		-1		1		4

You should produce a map where C maps to -2, E maps to -1, and A maps to 1. Explaining the E score this time, the answer of 4 to "I keep my thoughts to myself" adds 1 × (-1) = -1 to the E category, and the answer of 3 to "I show my gratitude" adds 0 × 1 = 0 to the E category for the net total of -1. None of the input questions assess the O or N factors, your resulting map should not have those as keys.

One last example. Suppose you have these question/response pairs:

Question	*Factors*					*Answer*
	O	***C***	***E***	***A***	***N***	
I reassure others.		-1		1		3
I feel others' emotions.			1	1		3

The resulting map should have C mappings to 0, E mapping to 0, and A mapping to 0. Exploring the A score here, the answer of 3 to "I reassure others" adds 0 × 1 = 0 to the A score, and the answer of 3 to "I feel others' emotions" adds a second 0 × 1 = 0 to the A score, for a net total of 0. Since none of the questions addressed factors O or N, you should leave those keys out.

***NOTE**: Although in the context of a personality quiz the keys in a map will be some subset of O, C, E, A, and N, your function should work even if we have other keys in the factors maps. (For example, we might repurpose this code to work with a different set of factors than OCEAN.)*

To summarize, here's what you need to do:

1. Implement the `scoresFrom` function in driver.h.
2. Use the provided tests to confirm that your code works correctly.
3. Test on your own by calling it in main.cpp and make your own tests to make sure it is working.

Milestone 3: Normalize Scores

We now have a map representing the user's OCEAN scores. Once we have these scores, we need to find character is "closest" to the user's scores. There are many ways to define "closest," but one of the most common ways to do this uses something called the cosine similarity, which is described below.

First, an observation. Suppose one person takes a personality quiz and answers ten questions, and a second, similar person takes a personality quiz and answers twenty questions. Suppose they get these scores:

	O	*C*	*E*	*A*	*N*
Person 1	6	-7	8	1	-3
Person 2	11	-14	17	2	-2

Notice that, generally speaking, the numbers for Person 2 have higher absolute value than the numbers for Person 1. There's a simple reason for this: since Person 2 answered more questions, they had more opportunities to have numbers added or subtracted from each category. This means that we'd expect the second person's numbers to have higher magnitudes than the first person's numbers.

To correct for this, you'll need to ***normalize*** the user's scores. Borrowing a technique from linear algebra, assuming the user's scores are o, c, e, a, and n, you should compute the value

$$\sqrt{o^2 + c^2 + e^2 + a^2 + n^2}$$

and then divide each of the user's five scores by this number. As an example, normalizing the two above scores gives these values:

	O	*C*	*E*	*A*	*N*
Person 1	0.476	-0.555	0.634	0.079	-0.238
Person 2	0.444	-0.565	0.686	0.081	-0.081

As you can see, the normalized scores of these two people are much closer to one another, indicating that their personalities are pretty similar. Your next task is to implement a function

```
map<char, double> normalize(map<char, int> &scores);
```

that takes as input a set of raw scores, then returns a normalized version of the scores.

There is an important edge case you need to handle. If the input map does not contain any nonzero values, then the calculation we've instructed you to do above would divide by zero. (Do you see why?) To address this, if the map doesn't contain at least one nonzero value, you should throw an error.

To summarize, here's what you need to do:

1. Implement the `normalize` function in driver.h.
2. Use the provided tests to confirm that your code works correctly.
3. Test on your own by calling it in main.cpp and make your own tests to make sure it is working.

NOTES:

- *There are no restrictions on what the keys in the map can be. Although you'll ultimately be using this function in the context of a personality quiz, you should not assume that the keys will be the letters O, C, E, A, and N. It might be that the keys are some subset of those five letters, or perhaps there will be other letters represented. In each case, sum up*

the squares of all the values, then divide each of the values by the square root of that sum.

- *The keys in the resulting map should be the same as the keys in the input map. In other words, you should not add or remove keys.*
- *Be mindful of the types involved in this function. The input map has integer keys representing whole-numbered scores. The output Map has keys that are arbitrary real numbers (hence the use of the type double).*
- *Remember that dividing two ints in C++ always results in an int. You might need to refresh your memory on C++'s hierarchy of types (see textbook).*

Milestone 4: Implement Cosine Similarity

At the end of the previous milestone, we saw this example of two similar OCEAN scores:

	O	***C***	***E***	***A***	***N***
Person 1	0.476	-0.555	0.634	0.079	-0.238
Person 2	0.444	-0.565	0.686	0.081	-0.081

We as humans can look at these numbers and say "yep, they're pretty close," but how could a computer do this? Suppose we have two different five-number scores (o_1, c_1, e_1, a_1, n_1) and (o_2, c_2, e_2, a_2, n_2) that have already been normalized. One measure we can use to determine how similar they are is to compute their ***cosine similarity***, which is defined here:

$$similarity = o_1o_2 + c_1c_2 + e_1e_2 + a_1a_2 + n_1n_2.$$

That is, you multiply the corresponding O scores, the corresponding C scores, the corresponding A scores, etc., then add them all together. The number you get back from the cosine similarity ranges from -1 to +1, inclusive. A similarity of -1 means "these people are about as diametrically opposite from one another as you could possibly get." A similarity of +1 means "these two people are as aligned as possible." If you run this calculation using the above numbers, you'll get a score of roughly 0.9855; these two people are remarkably similar!

Your next task is to write a function

```
double cosineSimilarityOf(const map<char, double> &lhs,
                          const map<char, double> &rhs);
```

that takes as input two sets of normalized scores, then returns their cosine similarity using the formula given above.

The *const* keywords ensures that neither parameter is changed after this function is called. The reason it's in the starter code is that without this keyword, these parameters are rather easy to change by accident. With this keyword, the C++ compiler will give you an error if you are using a non-const method (i.e., one that could potentially change the map) on the parameter.

To summarize, here's what you need to do:

1. Implement the `cosineSimilarityOf` function in driver.h.
2. Use the provided tests to confirm that your code works correctly.

3. Test on your own by calling it in main.cpp and make your own tests to make sure it is working.

***NOTES**:*

- *The two input maps are not guaranteed to have the same keys. For example, one map might have the keys A, C, and N, and the other might have the keys O and N. You should treat missing keys as if they're associated with zero weight.*
- *You can assume that the two sets of input scores are normalized and don't need to handle the case where this isn't true.*
- *As above, while in the finished product this code will be used on OCEAN scores, this function should work equally well if the keys are arbitrary characters.*

Milestone 5: Find the Best Match

At this point, you have the user's scores, and you have the ability to determine the similarity between pairs of scores. All that's left to do now is to tell the user which fictional character their scores are most similar to!

We've defined a type called Person in driver.h. It looks like this:

```
struct Person {
  string name;
  map<char, int> scores;
};
```

Here, name represents the person's name, and scores represents their five-factor OCEAN scores. (The scores are not normalized – can you tell why that is simply by looking at the type of the map?). Your final task is to write a function

```
Person mostSimilarTo(map<char, int> &scores, set<Person>& people);
```

that takes as input the user's raw OCEAN scores and a Set of fictional people. This function then returns the Person whose scores have the highest cosine similarity to the user's scores. As an edge case, if the people set is empty, you should throw an error, since there's no one to be similar to in that case. Similarly, if the user's scores can't be normalized – or if any of the people in the input set have scores that can't be normalized – you should throw an error.

To summarize, here's what you need to do:

1. Implement the `mostSimilarTo` function in driver.h.
2. Use the provided tests to confirm that your code works correctly.
3. Test on your own by calling it in main.cpp and make your own tests to make sure it is working.

NOTES:

- *Remember that you can only compute cosine similarities of normalized scores.*

- *If there is a tie between two or more people being the most similar, you can break the tie in whatever way you'd like.*
- *Take note the presence of the variable lowest_double at the top of driver.h, feel free to use this value to help find the maximum similarity.*
- *The most similar person to the user may have a negative similarity. As before, while we're ultimately going to be using this code on OCEAN scores, your function should work regardless of what the keys in the different maps are.*

Milestone 6: Write the user interface

Now that you have data processing framework developed and tested, it is time for you to write the app's user interface (all will be text based, but graphics would be fun to add on your own). Our app interface will be written in main.cpp. It will do all the file reading, user input/output, call all the functions in driver.h, etc. such that you have a fully functioning personality quiz app. Make sure you are writing this code using proper function decomposition, with maximum code reuse, with maximized efficiency, and with proper style.

You will have two file types to read in. The first is the questions, stored in questions.txt. Here is what the file looks like:

```
I am the life of the party. E:1
I talk to a lot of different people at parties. E:1
I start conversations. E:1
I love large parties. E:1
I don't talk a lot. E:-1
…
```

You will notice that each line will need to be parsed into a Question type (remember the struct from driver.h?).

You are also given a bunch of "people" files. For example, here is BabyAnimals.people:

```
Puppy. E:8 O:2 N:-3 C:-14 A:10
Kitty. E:-5 O:-4 A:-1 N:0 C:4
Bunny. A:14 N:0 E:3 O:2 C:-2
Baby Seal. A:9 C:-10 E:1 O:-1 N:-5
Baby Hamster. C:-1 O:0 E:9 N:0 A:-6
```

You will notice that each line will need to be parsed into a Person type (the second struct from driver.h). There are many other people files too, they are all formatted the same way.

Also, notice that the file formats are similar. We expect to see codes that utilize code reuse as much as possible.

Here is the sample input/output of your application (input in **red**):

```
Welcome to the Personality Quiz!

Choose number of questions: 3

How much do you agree with this
statement?
"I let others finish what they are
saying."

1. Strongly disagree
2. Disagree
3. Neutral
4. Agree
5. Strongly agree

Enter your answer here (1–5): 2

How much do you agree with this
statement?
"I am not afraid of providing
criticism."

1. Strongly disagree
2. Disagree
3. Neutral
4. Agree
5. Strongly agree

Enter your answer here (1–5): 4

How much do you agree with this
statement?
"I like to plan ahead."

1. Strongly disagree
2. Disagree
3. Neutral
4. Agree
5. Strongly agree

Enter your answer here (1–5): 5

1. BabyAnimals
2. Brooklyn99
3. Disney
4. Hogwarts
5. MyersBriggs
6. SesameStreet
7. StarWars
8. Vegetables
9. mine
0. To end program.

Choose test number (1–9, or 0 to
end): 1
You got Baby Hamster!

1. BabyAnimals
2. Brooklyn99
3. Disney
4. Hogwarts
5. MyersBriggs
6. SesameStreet
7. StarWars
8. Vegetables
9. mine
0. To end program.

Choose test number (1–9, or 0 to
end): 8
You got Green Bean!

1. BabyAnimals
2. Brooklyn99
3. Disney
4. Hogwarts
5. MyersBriggs
6. SesameStreet
7. StarWars
8. Vegetables
9. mine
0. To end program.

Choose test number (1–9, or 0 to
end): 0
Goodbye!
```

NOTE:

- *Do not worry about invalid inputs. Assume inputs given are within the correct ranges.*

Milestone 7: Creative Component

Part of the submission requirements are that you submit your own people file. You should come up with your own theme for a personality quiz and create a people file that can run in the personal quiz app. Check Mimir rubric for details on how it will be graded.

Optional: If you are interested in expanding this project into a personal project here are some ideas:

- Make a GUI. C++ is not the best language for this, but OpenGL is one possibility for graphics. You can find some starter code here. It is for XCode and may be outdated but might be enough to get you started. However, there are probably better ways to go about this and other languages and web tools are probably better.
- Once you have a five-factor score for someone, you can do all sorts of things. For example, what happens if you have five-factor scores for a bunch of different people? Could you play matchmaker (for any purpose…project partners, life partners, friends, etc.)?
- Could you explore ways in which different groups of people (students, professional bricklayers, airline pilots, etc.) are similar to or different from one another? Cosine similarities have uses way beyond personality quizzes. More generally, given a collection of data points, you can use cosine similarity as a way of determining how closely aligned they are. What other data sets could you apply this idea to?
- The five-factor model is only one of many different ways of computing a "personality score" for a person. Research some other way to do this, and see if you can design a better personality quiz than the one we gave here.

Requirements

1. You cannot change the function signatures in the starter code in driver.h. All the functions included are tested in unit tests in the autograder. You will not be able to pass the autograder if these are changed.
2. You are allowed to use and add other libraries (make sure to **include** them at the top of your file). However, you do not need many. Any standard C++ library is allowed for this project. You either must find them and figure out how to use them, or write your own functionality.
3. Each input file may be opened and read exactly once.
4. Your two code files (main.cpp and driver.h) must have a file header comment with your name and a program overview. Each function must have a header comment above the function (including the functions in driver.h), explaining the function's purpose, parameters, and return value (if any). Inline comments should be supplied as appropriate; comments like "*declares variable*" or "*increments counter*" are useless. Comments that explain non-obvious assumptions or behavior *are* appropriate.
5. No global variables; use parameter passing and function return. No heap allocation. No pointers.
6. The **cyclomatic complexity** (CCN) of any function should be minimized. In short, cyclomatic complexity is a representation of code complexity --- e.g.

nesting of loops, nesting of if-statements, etc. You should strive to keep code as simple as possible, which generally means encapsulating complexity within functions (and calling those function) instead of explicitly nesting code.

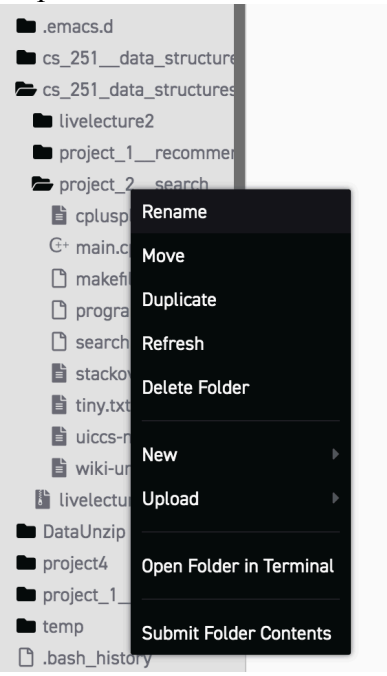
As a general principle, if we see code that has **more than 2 levels** of explicit looping --- an example of which is shown above --- that score will receive grade penalties. The solution is to move one or more loops into a function, and call the function.

Getting Started in Mimir

Please check Project 1 handout, scroll down to "Getting Started in Mimir" if you are really new to Mimir. Otherwise, I will be just added new Mimir information here.

How to submit files from my local computer to our submission system? This project requires you to submit multiple files, so here are two options for submitting files from your local machine to the Mimir submission system (if you are creating your .people file on a different computer and want to upload it).

(1) **Upload your file to Mimir IDE, submit from Mimir IDE**. To do this, you will first add files to a Mimir folder on the Mimir IDE. Navigate to the folder using the file directory GUI on the left panel of the Mimir IDE. Right click on the folder you'd like to upload the file to, click "Upload":



and then navigate to the file on your local computer that you'd like to upload. Looking at the file directory GUI, you should see the file appear or you can type ls in the terminal to see that file properly uploaded. Next, you will submit the Mimir folder contents by right clicking on the folder in Mimir and clicking "**Submit folder contents**". Follow the prompts to submit to the correct set of test cases. Make sure to preview your image in the submission folder to make sure it is viewable.

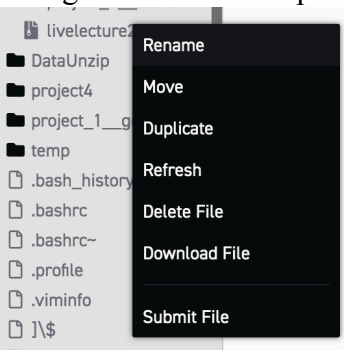
(2) Download your code from Mimir to your local machine, add any necessary files and submit to test cases from your computer. First, zip your Mimir folder by navigating one folder up from the folder you'd like to zip. Do this by "cd"-ing to the correct directory. You are in the right place if when you type "ls" you see the folder you want to zip. Here, I am going to zip up the folder livelecture2:

```
^_^[shanon] >>  ls
livelecture2  project_1__recommendations  project_2__search
```

Now type this command to zip the folder:

```
^_^[shanon] >>  ls
livelecture2  project_1__recommendations  project_2__search
^_^[shanon] >>  zip -r livelecture2.zip livelecture2/
  adding: livelecture2/ (stored 0%)
  adding: livelecture2/input2.txt (deflated 50%)
  adding: livelecture2/input.txt (deflated 28%)
  adding: livelecture2/main.cpp (deflated 62%)
  adding: livelecture2/makefile (deflated 32%)
  adding: livelecture2/mainSolution.cpp (deflated 62%)
  adding: livelecture2/program.exe (deflated 66%)
  adding: livelecture2/ourvector.h (deflated 74%)
^_^[shanon] >>
```

Navigate to the new .zip file in the file directory, right click on it, and click "Download File":

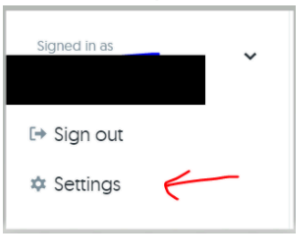


You can place this zipped file where ever you'd like on your computer and unzip. For most computer, double clicking on the file with unzip it. Now, you can add whatever files you'd like to the folder (like image file) and then submit back to Mimir. To do so, navigate to the Project 2 – Search homepage on Mimir and click "Submit":

The good part about this option is you have a backup of your work. Make sure you are backing up your work on Mimir regularly. You will lose access to Mimir at the end of the semester.

If you want to bulk backup your work on Mimir:

1.Go to the settings



2. On the top right corner you will see a button to download your IDE's contents and it creates a zip file with everything in your ide.

Please do this often. You will not have access to Mimir at the end of this course.

Citations/Resources

Assignment is credited to Keith Schwarz and Stuart Reges, Stanford University. Thanks to Kai Bonsol for his contributions.