CMPUT201 Assignment 6: Enums, Structs

- By: **YOUR_NAME_HERE**
- CCID: **YOUR_CCID_HERE**
- Student Number: **YOUR_STUDENT_NUMBER_HERE**

Sources

Tell us what online resources you used and who you collaborated with:

- **COLLABORATOR_1**
- **StackOverflow_Link**

Reminder: You may not use code from anyone else! Online resources and collaborators are for concepts only. As for all your assignments, this assignment will be checked for plagiarism using sophisticated tools so beware.

Goals

- Demonstrate knowledge of arrays
  - Fixed-Length Arrays
  - C99 Variable-Length Arrays
  - Bounds Checking
  - Memory layout
  - Multi-dimensional arrays
- Demonstrate knowledge of pointers
  - Difference between arrays and pointers
    - When does an array turn into a pointer
  - Unary `&` operator
  - Unary `*` operator
  - Subscripting a pointer `p[i]`
  - Modifying values "by reference"
  - Pointers are values
  - When are pointers valid?
- Demonstrate knowledge of malloc
  - When to allocate memory dynamically
  - Returning pointers pointing to arrays declared with malloc
- Demonstrate use of linters
  - Use linters to improve code quality

Code Quality Standards

Your code must meet the code quality standards. If you've taken CMPUT 174 before these should be familiar to you.

- Use readable indentation.
  - Blocks must be indented (everything between `{` and `}`)
  - One line must not have more than one statement on it. However, a long statement should be split into multiple lines.
- Use only idiomatic for loops.
- Use descriptive variable names. It must be obvious to the person reading (and marking your code) what each variable does.
- Never use complicated switch logic. Each case must fall through immediately to the next without running any code, or it must run some code and then break out of the switch statement.
- Never use goto.
- Never use control flow without curly braces (`if`, `else`, `do`, `while`, `for`, etc.)
- Use `<stdbool.h>`, `bool`, `true`, and `false` to represent boolean values.
  - Never compare with `true`, e.g. never `== true`.
- Do not leave commented-out code in your code.
- Provide comments for anything that's not totally and completely obvious.
- Always check to see if I/O functions were actually successful.
- On an unexpected error, print out a useful error message and exit the program.
  - For invalid input from the user you should handle it by asking the user to try again or by exiting the program with `exit(1)`, `exit(2)`, etc. or returning 1 or 2 etc. from main.
  - For unexpected errors, such as fgets failing to read anything, consider `abort()`.
- Main must only return 0 if the program was successful.
- Do not use magic literals (magic numbers or magic strings).
  - If a value has a particular meaning, give a meaningful name with `#define` or by declaring a constant with `const`.
  - Values other than 0 and 1 with the same meaning must not appear more than once.
  - 0 or 1 with a meaning other than the immediately obvious must also be given a name.
  - String literals must not appear more than once.
  - This includes magic numbers that appear in strings!
- Program must compile without warnings with `gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3`.
- Program must be architecture-independent:
  - Program must not rely on the sizes of int, long, size_t, or pointers.
  - Program must compile without warnings with `gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -m32`. Note the added **-m32**!
  - The result of this compilation must be an executable program.
  - The 32-bit program must produce the same output as the 64-bit program.

New Code Quality Standards

- **Program must be compiler-independent:**
  - Program must compile without warnings with `clang`.
    - You can use the same options for `clang` that you use for `gcc`!
  - Program compiled with `clang` should produce the same output as when it's compiled with `gcc`.
- **Code must be lint-free:**
  - Program must pass `clang-tidy --checks=*` without warnings, except those which are explicitly allowed.
    - Currently allowed:
      - `cert-err34-c`
      - `cert-msc30-c`
      - `cert-msc50-cpp`
    - More allowed warnings may be added. Check eClass for updates.
    - See instructions on how to run the linters below.
  - Program must pass `oclint` without warnings, except those which are explicitly allowed.
    - Currently allowed:
      - `UselessParentheses`
    - More allowed warnings may be added. Check eClass for updates.
    - See instructions on how to run the linters below.
- **Code must be well-organized into functions:**
  - Each function should do one thing and one thing only.
  - The function's name should indicate what it does.
  - The same code should never appear twice!
  - Functions should be short, simple, and take few parameters.
  - See "Organizing Code into Functions" on eClass under Guides and FAQs.
- **Code must use globals appropariately:**
  - Program must not use global mutable variables (variables without `const` outside of a function).
    - Program can use global constant variables (`const`).
    - Using constants with `const` is highly encouraged.
- **General:**
- Program must use `size_t` variables where appropriate.
- New types must be named in CamelCase (starting with a capital letter) or in all_lower_case_t ending with `_t`.
- Constants and defines must be named in ALL_CAPS.
- Mutable variables and functions must be named camelCase (starting with a lowercase letter) or in all_lower_case.
- Dynamically allocated memory shall be freed.

Testing your Program

Correct input-output examples are provided. For example, `q1a-test1-input.txt` is the input to your `./question1` program. If your program is correct, its output will match `q1a-test1-expected-output.txt`.

You can tell if your output matches exactly by saving the output of your program to a file with bash's `>` redirection operator. For example, `./question1 >my-output-1.txt` will save the output of your question1 program into the file named `my-output-1.txt` instead of showing it on the screen. Be warned! It will overwrite the file, deleting anything that used to be in `my-output-1.txt`.

Similarly, you can give input to your program from a file instead of typing it by using bash's `<` redirection operator. For example, `./question1 <q1a-test1-input.txt` will run your program with the contents of `q1a-test1-input.txt` instead of being typed out.

These two can be combined. For example,

```
./question1 <q1a-test1-input.txt >my-output-1.txt
```

will use the contents of `q1a-test1-input.txt` as input and save the output of your program in `my-output-1.txt`.

When you want to check if your output is correct, you can then use the `diff` command from bash to compare two files. For example,

```
diff -b my-output-1.txt q1a-test1-expected-output.txt
```

will compare the two files `my-output-1.txt` and `q1a-test1-expected-output.txt` and show you any differences. `-b` tells diff to ignore extra spaces and tabs.

`diff` will only show you something if there's a difference between the two files. If `diff` doesn't show you anything, that means the two files were the same!

So, putting it all together, to check if your program handles one example input correctly, you can run:

```
./question1 <q1a-test1-input.txt >my-output-1.txt
diff -b my-output-1.txt q1a-test1-expected-output.txt
```

If `diff` doesn't show you anything, that means the two files were the same, so your output is correct.

This is what the included scripts (`test-q1a.sh`, etc.) do.

However, the examples are just that: examples. If your code doesn't produce the correct output for other inputs it will still be marked wrong.

Linting Your Program

The two linters `clang-tidy` and `oclint` will examine your code for a HUGE number of problems.

For example:

- Long lines must be broken into short lines.
  - No line can be longer than 100 chars.
- Functions must be short.
  - No more than than 30 statements. (Check this with oclint, it will warn about "ncss" aka "non-commenting source statements").
- Functions must be simple.
  - Check this with oclint, it will warn about "complexity".
- All variables must be used.
- Don't leave any dead code.
  - Dead code is code that can never run.

Those are just a few of the things `clang-tidy` and `oclint` can check for. There are too many to list here. Because they check for so many things, we may find things that the linters think are problems that we don't think are really problems or that we don't have the tools to fix yet.

We will add them to the eClass list as we find them.

Running the Linters

Both linters take your C filename, some options, then a `--` followed by the exact way you would compile your code with `clang`.

The options for `clang-tidy` are currently `--checks=*,-cert-err34-c,-cert-msc30-c,-cert-msc50-cpp`, which tells `clang-tidy` to look for every problem, except the problems named `cert-err34-c`, `cert-msc30-c`, and `cert-msc50-cpp`.

The options for `oclint` are currently `--disable-rule=UselessParentheses`.

If we find more things that are allowed we will add them to these options.

For example, if you would compile your program with:

```
gcc -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o myprogram myprogram.c
```

then you could compile it with clang with:

```
clang -std=c99 -pedantic -Wall -Wextra -ftrapv -ggdb3 -o myprogram
myprogram.c
```

The only that changed was the name of the compiler. So you would run clang-tidy and oclint like:

```
clang-tidy --checks=*,-cert-err34-c myprogram.c -- -std=c99 -pedantic -Wall -
Wextra -ftrapv -ggdb3 -o myprogram myprogram.c
oclint --disable-rule=UselessParentheses myprogram.c -- -std=c99 -pedantic -
Wall -Wextra -ftrapv -ggdb3 -o myprogram myprogram.c
```

Notice that you have to specify `myprogram.c` twice. This is because `oclint` and `clang-tidy` need to know both what file you want them to look at and exactly how you would compile it.

clang, clang-tidy, and oclint aren't on the lab machines :(This is due to Campus IT (IST) not keeping the lab machine's OS up to date. Please use the VM if at all possible. If absolutely can't run the VM, check back and we will have a way for you to run them soon.

Hints

- Solve "complexity" warnings by splitting your code into more functions.
    - Instead of putting a bunch of code inside of an loop, just call a function.
    - Instead of putting a bunch of code inside of an `if`, call a function.
- Don't use isdigit, etc. (man 3 isdigit) They cause linter warnings.
- Breaking up long lines.
    - Remember, C doesn't care too much about whitespace, so you can spread your statement over multiple lines.
    - Just be sure to use indentation to make it clear what you are doing.

```
\\ Example 1:
r = a + b * c + d * e * f;
\\ You can rewite it as...
r = a
    + b * c
    + d * e * f;

\\ Example 2:
if (a == b && c == d && e == f) { }
\\ You can rewrite it as...
if (
    a == b
    && c == d
    && e == f
) {
}
```

Questions

Question 1

Overview

The user is going to provide you with a map of rivers and grassland. Each cell on the map will be either grassland or river. Your job is to decorate this map with forks, 4 way forks, and river

bends when a river bends. You will process the user's map and modify the river cells if they depict a river bend.

You will use enums to model the river, its bends, and forks. You will pattern match to replace parts of the river with bends and forks. Out of bounds regions will be considered grasslands for pattern matching simplicity.

You will use enums in this program.

You will print the integers with 1 space padding and rows will be terminated by new lines:

```
10 10 10
 1  2  3
 4  5  6
 7  8  9
10 10 10
```

Input and Output

The tiles that we use in the map are:

- 0 Grassland
- 1 River
- 2 NorthWestRiverBend
- 3 SouthWestRiverBend
- 4 NorthEastRiverBend
- 5 SouthEastRiverBend
- 6 NorthEastSouthFork
- 7 NorthWestSouthFork
- 8 WestNorthEastFork
- 9 WestSouthEastFork
- 10 FourWayFork

The user will input maps usually of 0 and 1 but they can include other tiles as well. Typically it will 0 and 1.

The user will input a map of

```
P2
6 7
10
 0  0  0  0  0  0
 0  1  1  1  1  0
 0  1  0  0  1  0
 0  1  1  1  1  1
 0  1  0  0  1  0
 0  1  1  1  1  0
 0  0  0  0  1  0
```

Where 0 is grassland and 1 is river

The program will look for corners and replace them with the RiverBend pieces.

```
P2
6 7
10
 0  0  0  0  0  0
 0  5  1  1  3  0
 0  1  0  0  1  0
 0  6  1  1 10  1
 0  1  0  0  1  0
 0  4  1  1  7  0
 0  0  0  0  1  0
```

The header format is described below. $WIDTH is the number of cells wide and $HEIGHT is the number of cells tall.

```
P2
$WIDTH $HEIGHT
10
```

The individual cells are whitespace seperated for input. Ignore whitespace and newlines for the cells.

The output format is strict, each row ends in a newline and there is 1 space padding for all integers cells printed.

Please review the q1a-test?-input.txt files for more examples

Patterns

- Examples of No change

```
 0  0  0          0  0  0
 0  0  0    ->    0  0  0
 0  0  0          0  0  0

 0  0  0          0  0  0
 0  0  0    ->    0  0  0
 0  0  1          0  0  1

 0  0  0          0  0  0
 0  0  0    ->    0  0  0
 1  1  1          1  1  1
```

These examples are in cross form, this means that the corners can be anything (river or grassland). This is only 1 cell being changed, you are expected to apply these patterns to all cells.

- 2 NorthWestRiverBend

```
    1                   1
 1  1  0     ->    1   2   0
    0                   0
```

For example:

```
 0  1  0           0   1   0
 1  1  0     ->    1   2   0
 0  0  0           0   0   0
```

- 3 SouthWestRiverBend

```
    0                   0
 1  1  0     ->    1   3   0
    1                   1
```

For example:

```
 1  0  1           1   0   1
 1  1  0     ->    1   3   0
 1  1  1           1   1   1
```

- 4 NorthEastRiverBend

```
    1                   1
 0  1  1     ->    0   4   1
    0                   0
```

- 5 SouthEastRiverBend

```
    0                   0
 0  1  1     ->    0   5   1
    1                   1
```

- 6 NorthEastSouthFork

```
    1                   1
 0  1  1     ->    0   6   1
    1                   1
```

- 7 NorthWestSouthFork

```
    1                   1
 1  1  0     ->    1   7   0
    1                   1
```

- 8 WestNorthEastFork

```
    1                   1
```

```
 1  1  1     ->    1  8  1
    0                0
```

- 9 WestSouthEastFork

```
    0                0
 1  1  1     ->    1  9  1
    1                1
```

- 10 FourWayFork

```
    1                1
 1  1  1     ->    1 10  1
    1                1
```

More details

Maximum supported dimension in width or height is 16000

The format you input and output is called plain PGM. PGM is portable grey map format so you can use some image programs to view it.

Invalid input (including unexpected EOF) should be aborted immediately with the message:

```
Invalid input!
```

For examples, check the tar file.

We provide `question1.sh`, `question1-clang.sh`, and `question1-lint.sh` in the tar file.

Additional Requirements

- Put your C code for this question in `question1.c`
- You should compile the program as `./question1`
- You must demonstrate the proper use of functions calls and defining functions.
- You must not use global variables or static local variables, unless they are constants declared with `const`.
- You may ignore extra input.
- You may abort on any invalid input.
- You may use scanf for input.
- You may not use global variables (except constants with `const`).

Marking

- **1 Point** Program is well-organized into functions. (See above.)
- **1 Point** Program uses enums appropriately. (See above.)

- **1 Point** `question1.c` and `question1.sh` meets the requirements above and program output is correct for a valid input. (Examples: `test-q1a.sh`)
- **1 Point** `question1.c` and `question1.sh` meets the requirements above and program output is correct for a invalid input. (Examples: `test-q1b.sh`)
- **1 Point** `question1.c` and `question1-clang.sh` meets the requirements above, and your program is *compiler*-independent as described above. (Examples: `test-q1a-clang.sh` `test-q1b-clang.sh`)
- **1 Point** `question1-lint.sh` runs both linters correctly, as above, and your program is lint-free.
- **1 Point** Quality of question1.c meets all other quality standards, listed above.

Hints

- Initialize your memory when you malloc it
- Remember to check the bounds of the array.
- You should make/extract a cursor so you can pattern match.
- You should consider writing unit tests for the patterns
- You should consider creating new arrays dynamically where need be.
- Return appropriate values from the functions to avoid segmentation faults.
- If you find any task repetitive, make it a function.
- You should read from 1 2D array and write to another.
- You can visualize your output better if you pipe your program through utfdraw.sh (you need GNU sed)
- You can modify your assignment 4 solution.

Question2

In this question you have to implement a new data structure called Squeue. The queue behaves a bit like both a Stack and a Queue, that allows new elements to be added and removed at both ends of the list. A Squeue supports the following operations: **initSqueue, addFront, addBack, reverseSqueue** and **printSqueue**.

You will implement the Squeue by using nodes. Each node contains a *next* and a *prev* pointer and therefore it is linked to the nodes before and after it in the Squeue. The Squeue may contain one or many such nodes but the important thing to note is that you know where Squeue begins and ends. This information is kept track of using two special pointers which are part of Squeue, namely *first* and *last*. The *first* points to the first element of the Squeue, whereas the *last* points to the last element of Squeue. The formal definition of the structs is shown in the following:

```
struct node {
  char *value;
  struct node *next;
  struct node *prev;
};
```

```
typedef struct node Node;


struct squeue{
   struct node* first;
   struct node* last;
};
typedef struct squeue *Squeue;
```

Apart from the above two structs you are required to use the direction enum shown below. This will help you in printing the Squeue in the forward or backward direction.

```
enum direction {
    FORWARD,
    BACKWARD
};
typedef enum direction Direction;
```

Now having an overall idea of the Squeue you are required to implement the basic functions for Squeue. Below we provide the details of what each function is expected to do. In addition to this you will be given some code in question2.c file that you can use to begin your program.

initSqueue:

```
/* initSqueue: initialize the given squeue to an empty squeue by allocating
memory
 * using malloc. The squeue may not hold any data at this point. The function
basically
 * prepares the squeue for the next operations i.e., adding and printing
elements etc.
 * @returns the new malloc'd Squeue
 */
Squeue initSqueue() {
```

freeSqueue

```
/* freeSqueue: frees the squeue, its nodes, and the strings contained by the
nodes.
 *
 * @param squeue -- the squeue to free. All nodes and all of their values
will be free'd as well.
 */
void freeSqueue(Squeue squeue) {
```

addFrontSqueue:

```
/* addFrontSqueue: adds a node with a value to the front of squeue. The
 * pointer to the squeue where value will be added and the value
 * itself are provided by the caller function. The function adds the
 * node with the copied value to the front of given squeue and adjusts
 * all the pointers accordingly. If the given value is the first value
 * to be added then *first* and *last* pointer in squeue shall point
 * to this newly added value after the function completes its
```

```
 * execution. If this value is not the first value update the pointers
 * appropriately so that the every element in the squeue is linked.
 * This means you will have to update the *next* pointer of this newly
 * added node and the *prev* pointer of the node that was at the front
 * of squeue before this. You will also update *first* pointer so that
 * it points to this new node now.
 *
 *
 * value will be copied and malloc'd (you can strndup)
 *
 * @param squeue, a pointer to the struct Squeue
 * @param val, a pointer to a char
 * @returns void
 */
void addFrontSqueue(Squeue squeue, char* value) {
```

addBackSqueue:

```
/* addBackSqueue: adds a value to the back(or end) of squeue. The pointer to
the squeue in
 * which the value will be added and the value itself are provided by the
caller
 * function. The function adds the value to the end of the given squeue and
adjusts
 * all the pointers accordingly. If the given value is the first value to be
added
 * to the squeue then *first* and *last* pointer in squeue shall point to
this newly
 * added value  after the function completes its execution. If this value is
not the
 * first value then update the pointers appropriately so that the every
element in the
 * squeue is linked. This means you will have to update the *prev* pointer of
this newly
 * added node and the *next* pointer of the node that was at the end of the
squeue before
 * this. You will also update *last* pointer so that it points to this new
node now.
 *
 * value will be copied and malloc'd (you can strndup)
 *
 * @param squeue, a pointer to the struct Squeue
 * @param val, a pointer to a char
 * @return void
 */
void addBackSqueue( Squeue squeue, char* value) {
```

printSqueue:

```
/* printSqueue: prints the squeue in either forward or backward direction as
indicated by
 * the argument passed to the function. If the direction passed is FORWARD
print the values
 * of each node traversing squeue from the *first* node to the *last* node.
If the direction
```

```
 * passed is BACKWARD then start traversal and printing from the *last*
element of squeue until
 * you reach *first* node.
 *
 * @param squeue, a pointer to the struct squeue
 * @param direction, an enum representing either the FORWARD or the BACKWARD
direction
 * @return void
 */
void printSqueue( const struct squeue * squeue, Direction dir) {
```

reverseSqueue:

```
/* reverseSqueue: reverses the nodes in the squeue such that the *last*
pointer now points
 * to the *first* node and *first* points to the *last* node. For example, if
the squeue
 * was a->b->c->d , where *first* points to 'a' and *last* points to 'd' ,
calling reverse
 * would change the squeue contents to d->c->b->a , and make the first point
to 'd' and
 * last point to 'a' . The *next* and *prev* of all nodes will also be
changed accordingly.
 *
 * @param squeue, a pointer to the struct squeue
 * @return void
 */
void reverseSqueue(Squeue squeue) {
```

You are provided with a driver (main) and the example tests to help you run the program in an interactive way. Fill in the provided functions and do not change prototypes.

Additional Requirements

- Put your C code for this question in `question2.c`
- You should compile the program as `./question2`
- You must demonstrate the proper use of functions calls and defining functions.
- You must not use global variables or static local variables, unless they are constants declared with `const`.
- You may abort on any invalid input.
- You may use scanf for input.
- You may not use global variables (except constants with `const`).

Marking

- **1 Point** Program is well-organized into functions. (See above.)
- **1 Point** `question2.c` and `question2.sh` meets the requirements above and program output is correct for a valid input. (Examples: `test-q2a.sh`)

- **1 Point** `question2.c` and `question2.sh` meets the requirements above and program output is correct for a invalid input. (Examples: `test-q2b.sh`)
- **1 Point** `question2.c` and `question2-clang.sh` meets the requirements above, and your program is *compiler*-independent as described above. (Examples: `test-q2a-clang.sh` `test-q2b-clang.sh`)
- **1 Point** `question2-lint.sh` runs both linters correctly, as above, and your program is lint-free.
- **1 Point** Quality of question2.c meets all other quality standards, listed above.

Hints

- Remember to check for NULL pointers
- Remember to check the end of array using null character.
- Return appropriate values from the functions to avoid segmentation faults.
- If you find any task repetitive, make it a function.
- Remember to free your malloc'd values.
- Read the function void heavilyExerciseSqueue()

Submission

Test your program!

Always test your code on the VM or a Lab computer before submitting!

You can assume the shell script is run in the directory that contains both the source code and the executable. Run the `test-q1a.sh` script for question1. Run the `test-q1b.sh` script for question1. Run the `test-q2a.sh` script for question2. Run the `test-q2b.sh` script for question2. The scripts should produce no output.

Test your program with clang and lint your program

Unfortunately `clang` and the linters aren't available on the lab machines, so you need to use the VM for this step. If you aboslutely cannot use the VM, please wait a couple of days and we will have a solution for you.

Make 1 line (excluding the comments and header) shell scripts for question 1 and question 2 that will compile and run the 64-bit C program for that question with `clang`. Name the scripts `question1-clang.sh` and `question2-clang.sh` respectively. Run the `test-q1a-clang.sh` script for question1. Run the `test-q1b-clang.sh` script for question1. Run the `test-q2a-clang.sh` script for question2. Run the `test-q2b-clang.sh` script for question2.

Lint your program!

Hint: check `question2-lint.sh` for an example.

Run the `question2-lint.sh` script for question2. It's in the example tar. Run the `question1-lint.sh` script for question1. It's in the example tar.

To lint and check the code of your questions. If there are warnings, fix the code and try again.

Tar it up!

Make a tar ball of your assignment. It must not be compressed. The tar name is `__YOUR__CCID__-assignment6.tar`

the tar ball should contain:

- `__YOUR__CCID__-assignment6/` # the directory
- `__YOUR__CCID__-assignment6/README.md` # this README filled out with your name, CCID, ID #, collaborators and sources.
- `__YOUR__CCID__-assignment6/question1.c` # C program
- `__YOUR__CCID__-assignment6/question2.c` # C program
- `__YOUR__CCID__-assignment6/question1` # executable
- `__YOUR__CCID__-assignment6/question2` # executable
- `__YOUR__CCID__-assignment6/question1.sh` # shell script
- `__YOUR__CCID__-assignment6/question1-clang.sh` # shell script
- `__YOUR__CCID__-assignment6/question1-lint.sh` # shell script
- `__YOUR__CCID__-assignment6/question2.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment6/question2-clang.sh` # shell script -- should be exactly the same as the example
- `__YOUR__CCID__-assignment6/question2-lint.sh` # shell script -- should be exactly the same as the example

Extra files such as the test files are allowed to be in the tar file. Any file we provide you in the release tar is OK to be in your tar file.

Submit it!

Upload to eClass! Be sure to submit it to the correct section.

Marking

This is a 13-point assignment. It will be scaled to 4 marks. (4% of your final grade in the course: A 13/13 is 100% is 4 marks.) Partial marks may be given at the TA's discretion.

- You will lose all marks if not a tar (a `.tar` file that can be unpacked using `tar -xf`)

- You will lose all marks if files not named correctly and inside a correctly named directory (folder)
- You will lose all marks if your C code is not indented. Minor indentation errors will not cost you all your marks.
- You will lose all marks if your code does not compile on the VMs or the lab machines.
- You will lose all marks if `README.md` does not contain the correct information! Use our example README!
    - Markdown format (use `README.md` in the example as a template)
    - Name, CCID, ID #
    - Your sources
    - Who you consulted with
    - The license statement below

License