<u>Exercise #1: Threads</u>

Examine the code in:

```
./test/exercise_01_threads/test__exercise_01_threads.cc
```

What do you think it will output when run?

I believe that when this code is run, it will output the letters ‘a’, ‘b’, and ‘c’ in an almost random-like sequence. This is due to the scheduling of the threads.

Build the repo. From the `./build` directory, run:

```
./test/exercise_01_threads/test__exercise_01_threads
```

Does the output match your expectations? Run the program several more times. What do you notice?

Yes it does, we see how the letters are randomized. We notice how the output is different each time.

When execution reaches `LINE A`, how many threads are running?

When execution reaches LINE A, there are 4 threads running.

Comment out one of the `join()` lines, recompile, and rerun the program. What happened? Why? Run `ls` in the `./build` directory. What are those new files? Uncomment the line.

There is an error. This is because we didn’t there are 3 initialized threads, but we are only joining 2 of them. In other words, you are not allowed to not join a thread.

Assuming that we don’t care about the exact ordering of the letters, are there any problems with this code?

Yes, there is a chance for a race condition to occur if multiple threads write to cout at the same time as there is no locking mechanism in place.

<u>Exercise #2: Race Conditions</u>

Examine the code in:

```
./test/exercise_02_rcs/test__exercise_02_rcs.cc
```

Why are `counter` and `counter_mutex` being passed by reference to the threads?

‘Counter’ and ‘counter_mutex’ are being passed by reference because if we did it by value, each thread would be operating with different copies and the final increment would not be the same. Passing by reference ensures that all threads work on the same counter address, and therefore give us an accurate final increment.

How long (approximately) will it take the test to run? Will it pass?

About 1 second. It doesn’t pass.

From the `./build` directory, run:

```
time ./test/exercise_02_rcs/test__exercise_02_rcs
```

Do the outputs and times match your previous expectations? Run the command several times.

Output and times do match my previous expectations.

In `increment_counter()`, insert a `lock_guard` inside the `for` loop to protect the increment operation. Rebuild and rerun the code. Does the test pass? How long does it take to run?

Yes, the test passes now. It takes around 3 seconds to run.

Put the lock_guard and increment operations (only) inside a local scope block. Recompile and rerun the code. Does the test pass? How long does it take to run? What does this say about which code should be inside critical sections?

The test indeed does pass. It takes about 1 second to run. Code that accesses shared resources should be inside critical sections as by reducing the time the mutex is locked, we can speed up how long it takes the code to run.

<u>Exercise #3: Deadlock</u>

Examine the code in:

```
./test/exercise_03_deadlock/test__exercise_03_deadlock.cc
```

Are there any concurrency issues in this code?

Yes, we have 2 threads locking each other, preventing the code from going forward.

From the `./build` directory, run:

```
./test/exercise_03_deadlock/test__exercise_03_deadlock
```

What happens? Why?

The code does not run. It takes having to quit the program for the failed test cases to show. This is because in the code, we have a deadlock, where 2 threads are locking each other.