- Exercises

In this exercise, you will use locks instead of mutexes.

WE'LL COVER THE FOLLOWING

Task 1

- Adjust the program below by using a suitable lock: `std::unique_lock` or `std::lock_guard`.

> 💡 **You should not explicitly use a mutex.**

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <string>
#include <thread>

std::mutex coutMutex;

class Worker{
public:
  explicit Worker(const std::string& n):name(n){};

    void operator() (){
      for (int i= 1; i <= 3; ++i){
            // begin work
            std::this_thread::sleep_for(std::chrono::milliseconds(200));
            // end work
            //coutMutex.lock();
            std::cout << name << ": " << "Work " << i << " done !!!" << std::endl;
            //coutMutex.unlock();
```

```
      }
        }
private:
  std::string name;
};


int main(){

  std::cout << std::endl;

  std::cout << "Boss: Let's start working." << "\n\n";

  std::thread herb= std::thread(Worker("Herb"));
  std::thread andrei= std::thread(Worker("  Andrei"));
  std::thread scott= std::thread(Worker("    Scott"));
  std::thread bjarne= std::thread(Worker("      Bjarne"));
  std::thread andrew= std::thread(Worker("        Andrew"));
  std::thread david= std::thread(Worker("          David"));

  herb.join();
  andrei.join();
  scott.join();
  bjarne.join();
  andrew.join();
  david.join();

  std::cout << "\n" << "Boss: Let's go home." << std::endl;

  std::cout << std::endl;

}
```

Task 2

Implement a count-down counter from 10 – 0. It should count down in seconds steps.

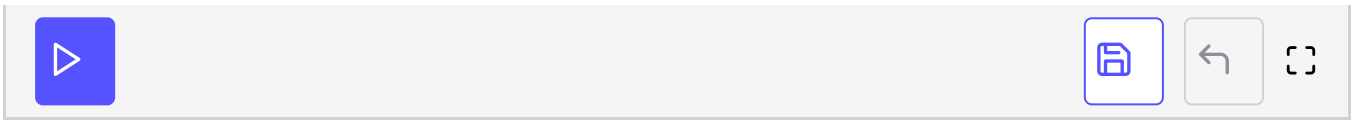
```
#include <iostream>

int main() {
  // your code goes here
  std::cout << "Hello World";
  return 0;
}
```

Try It Out!

Study the code below along with its explanation before you try out a few things on your own.

```
#include <chrono>
#include <iostream>
#include <map>
#include <mutex>
#include <shared_mutex>
#include <string>
#include <thread>

std::map<std::string, int> teleBook{{"Dijkstra", 1972}, {"Scott", 1976}, {"Ritchie", 1983}};

std::shared_timed_mutex teleBookMutex;

void addToTeleBook(const std::string& na, int tele){
  std::lock_guard<std::shared_timed_mutex> writerLock(teleBookMutex);
  std::cout << "\nSTARTING UPDATE " << na;
  std::this_thread::sleep_for(std::chrono::milliseconds(500));
  teleBook[na]= tele;
  std::cout << " ... ENDING UPDATE " << na << std::endl;
}

void printNumber(const std::string& na){
  std::shared_lock<std::shared_timed_mutex> readerLock(teleBookMutex);
  std::cout << na << ": " << teleBook[na] << std::endl;
}

int main(){

  std::cout << std::endl;

  std::thread reader1([]{ printNumber("Scott"); });
  std::thread reader2([]{ printNumber("Ritchie"); });
  std::thread w1([]{ addToTeleBook("Scott", 1968); });
  std::thread reader3([]{ printNumber("Dijkstra"); });
  std::thread reader4([]{ printNumber("Scott"); });
  std::thread w2([]{ addToTeleBook("Bjarne", 1965); });
  std::thread reader5([]{ printNumber("Scott"); });
  std::thread reader6([]{ printNumber("Ritchie"); });
  std::thread reader7([]{ printNumber("Scott"); });
  std::thread reader8([]{ printNumber("Bjarne"); });

  reader1.join();
  reader2.join();
  reader3.join();
  reader4.join();
  reader5.join();
  reader6.join();
  reader7.join();
  reader8.join();
  w1.join();
  w2.join();

  std::cout << std::endl;
```

```
  std::cout << "\nThe new telephone book" << std::endl;
  for (auto teleIt: teleBook){
    std::cout << teleIt.first << ": " << teleIt.second << std::endl;

  }

  std::cout << std::endl;

}
```



Explanation

- The `telebook` in line 9 is the shared variable that must be protected.
- Eight threads want to read the telephone book. Two threads want to modify (lines 30 - 39) it.
- To access the telephone book simultaneously, the reading threads use the `std::shared_lock<std::shared_timed_mutex>>` in line 22.
- This is opposite to the writing threads, which need exclusive access to the critical section. The exclusivity is given by the `std::lock_guard<std::shared_timed_mutex>>` in line 14.
- At the end, the program displays the updated telephone book in lines 54 – 57.

Your Turn!

The program has a data race on the telephone book. The program has a data race on the telephone book. The issue with the telephone book is that the call `teleBook[na]` in line 23 could be a write operation when `na` is not in the telephone book. Since line 23 is performed without synchronization, this is a data race. To test the problem, make line 39 to the first thread, and after a few executions, you will see that `Bjarne` returns as the value 0.

The semantic of the map's index operator states that the default value will be created if the key is not available (see here). If one interleaving threads has a data race, the program has undefined behavior. Less than 5 % of the developers can spot this data race. Thread sanitizer will not help in these situations since an `std::map` is a highly complicated structure.

The solutions to the above tasks can be found in the next lesson.