Types of Locks: std::shared_lock

This lesson gives an overview of the std::shared_lock which is a type of lock used in C++.

A `std::shared_lock` has the same interface as a `std::unique_lock` but behaves differently when used with a `std::shared_timed_mutex`. Many threads can share one `std::shared_timed_mutex` and, therefore, implement a reader-writer lock. The idea of reader-writer locks is straightforward and extremely useful. An arbitrary number of threads executing read operations can access the critical region at the same time, but only one thread is allowed to write.

Reader-writer locks do not solve the fundamental problem - threads competing for access to a critical region, but they do help to minimize the bottleneck.

Telephone book example for reader-writer locks

A telephone book is a typical example using a reader-writer lock. Usually, a lot of people want to look up a telephone number, but only a few want to change them. Let's look at an example.

```
// readerWriterLock.cpp

#include <iostream>
#include <map>
#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];
}

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;

}
```

The telephone book in line 9 is the shared variable, which has to be protected

The telephone book in line 9 is the shared variable, which has to be protected. Eight threads want to read the telephone book, two threads want to modify it

(lines 31 - 40). To access the telephone book at the same time, the reading threads use the `std::shared_lock<std::shared_timed_mutex>>` in line 23. This is in contrast 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 15. In the end, the program displays the updated telephone book (lines 55 - 58). The output of the reading threads overlaps, while the writing threads are executed one after the other. This means that the reading operations are performed at the same time. That was easy. Too easy. The telephone book has undefined behavior

Undefined Behaviour

The program has undefined behavior. To be more precise it has a data race. What? Before you continue, stop for a few seconds and think. By the way the concurrent access to `std::cout` is not the issue.

The characteristic of a data race is that at least two threads access the shared variable at the same time and at least one of them is a writer. This exact scenario may occur during program execution. One of the features of the ordered associative container is that reading of the container can modify it. This happens if the element is not available in the container. If “Bjarne” is not found in the telephone book, a pair (“Bjarne”,0) will be created from the read access. You can simply force the data race by putting the printing of Bjarne in line 40 in front of all the threads (lines 31 - 40). Let’s have a look.

You can see it right at the top, Bjarne has the value 0.

```
// readerWriterLock.cpp

#include <iostream>
#include <map>
#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){
```

```
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];
}

int main(){

  std::cout << std::endl;

  std::thread reader8([]{ printNumber("Bjarne"); });
  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"); });


  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;

}
```

An obvious way to fix this issue is to use only reading operations in the function `printNumber`:

```
// readerWriterLock.cpp
// readerWriterLock.cpp

#include <iostream>
#include <map>
#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);
  auto searchEntry = teleBook.find(na);
  if(searchEntry != teleBook.end()){
    std::cout << searchEntry->first << ": " << searchEntry->second << std::endl;
  }
  else {
    std::cout << na << " not found!" << std::endl;
  }
}


int main(){

  std::cout << std::endl;

  std::thread reader8([]{ printNumber("Bjarne"); });
  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"); });

  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;

}
```

If a key is not in the telephone book, I will simply write `not found` to the console.

You can see the message `Bjarne not found!` in the output of the second program execution. In the first program execution, `addToTeleBook` will be executed first; therefore, Bjarne will be found.