Arguments of Threads: Race Conditions and Locks

This lesson defines race conditions, and explains how to address them in concurrent programming with C++.

Both issues from the previous lesson are actually race conditions because the result of the program depends on interleaving the operations. The race condition is the cause of the data race.

Fixing the data race is quite easy; `valSleeper` should be protected using either a lock or an atomic. To overcome the lifetime issues of `valSleeper` and `std::cout`, we have to join the thread instead of detaching it.

Here is the modified `main` function:

```cpp
#include <chrono>
#include <iostream>
#include <thread>

class Sleeper{
  public:
    Sleeper(int& i_):i{i_}{};
    void operator() (int k){
      for (unsigned int j= 0; j <= 5; ++j){
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        i += k;
      }
      std::cout << std::this_thread::get_id() << std::endl;
    }
  private:
    int& i;
};

int main(){

  std::cout << std::endl;

  int valSleeper= 1000;
  std::thread t(Sleeper(valSleeper),5);
  t.join();
  std::cout << "valSleeper = " << valSleeper << std::endl;

  std::cout << std::endl;

}
```

Now we get the right result; of course, the execution becomes slower.

In the next lesson, we’ll solve an exercise related to the above example.