- Solutions

The solutions to the exercises in the previous lesson. Here you can test the techniques you learned.

WE'LL COVER THE FOLLOWING

- Solution 1
  - Explanation
  - Try It Out!
- Solution 2

Solution 1

```
//threadHardwareConcurrency.cpp
#include <chrono>
#include <iostream>
#include <thread>

class Sleeper{

  public:
    Sleeper(int& i_, int m):i{i_}, milli(m){};
    void operator() (int k){
      for (unsigned int j= 0; j <= 5; ++j){
        std::this_thread::sleep_for(std::chrono::milliseconds(milli));
        i += k;
      }
    }
  private:
    int& i;
    int milli;
};


int main(){

  std::cout << std::endl;

  for (unsigned int i=0; i <= 20; ++i){

    int valSleeper= 1000;
    std::thread t(Sleeper(valSleeper, (i*50)), 5);
    t.detach();
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

```
    std::cout << "valSleeper = " << valSleeper << std::endl;

  }

  std::cout << std::endl;

}
```




Explanation

- Although the solution seems simple, it has one significant issue.
- There is a data race on the mutable shared variable `valSleeper`, since a read (line 32) and write (line 13) occurs simultaneously.

Try It Out!

Let the program run with the thread sanitizer in order to visualize the data race. Try the following command line below `g++ -std=c++14 -fsanitize=thread -pthread -g program.cpp -o program -p -lpthread`. Compare the output of the program below with the solution above. Take a look at the hint if you are unable to run the file.


Show Hint

Terminal


Solution 2

```
// threadHardwareConcurrency.cpp
#include <iostream>
#include <thread>

int main(){

  std::cout << "std::thread::hardware_concurrency(): " << std::thread::hardware_concurrency()

}
```




For further information, see threads.

In the next lesson, we will study mutexes.