One of the biggest challenges of thread-management begins when the threads share non-const data

Data race and critical section

**Data Race**

    A data race is a state, in which at least two threads access a shared data at the same time, and at least one of the threads is a writer.

**Critical Section**

    A critical section is a section of the code, which not more than one thread should access at any point in time.

A nice way to visualize a race condition is to let a few threads write to std::cout. std::cout is the shared object (output stream), that should be protected from simultaneous access by multiple threads.

// coutUnsynchronized.cpp

#include <chrono>

#include <iostream>

#include <thread>

class Worker{

public:

Worker(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

std::cout << name << ": " << "Work " << i << " done !!!" << std::endl;

}

}

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;

}

### A side note: std::cout is thread safe

The C++11 standard guarantees, that you must not protect the single characters, written to std::cout. Each character will atomically be written. Of course, it is possible, that more output statements like in the example will interleave. But that is only an optical issue. The program is well defined. The remark is valid for all input and output streams.

## Mutex

Mutex stands for **mut**ual**ex**clusion. It ensures, that only one thread can access a critical section.

// coutSynchronized.cpp

#include <chrono>

#include <iostream>

#include <mutex>

#include <thread>

std::mutex coutMutex;

class Worker{

public:

Worker(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;