Promise and Future: Return a Notification

This lesson teaches how to return a notification while using std::promise and future in C++ for multithreading.

If you use promises and futures to synchronize threads, they have a lot in common with condition variables. Most of the time, promises and futures are the better choices. Before I present you an example, here is the big picture.

Criteria	Condition Variables	Tasks
Multiple synchronizations	Yes	No
Critical section	Yes	No
Error handling in receiver	No	Yes
Spurious wakeup	Yes	No
Lost wakeup	Yes	No

The advantage of a condition variable to a promise and future is that you can use condition variables to synchronize threads multiple times. In contrast to that, a promise can send its notification only once, so you have to use more promise and future pairs to get the functionality of a condition variable. If you use the condition variable for only one synchronization, the condition variable is a lot more difficult to use in the right way. A promise and future pair needs no shared variable and, therefore, it doesn't have a lock, and isn't

prone to spurious or lost wakeups. In addition to that, tasks can handle exceptions. There are lots of reasons to prefer tasks to condition variables.

Do you remember how difficult it was to use condition variables? If not, here are the key parts required to synchronize two threads.

```
void waitingForWork(){
    std::cout << "Worker: Waiting for work." << std::endl;

    std::unique_lock<std::mutex> lck(mutex_);
    condVar.wait(lck, []{ return dataReady; });
    doTheWork();
    std::cout << "Work done." << std::endl;
}

void setDataReady(){
    std::lock_guard<std::mutex> lck(mutex_);
    dataReady=true;
    std::cout << "Sender: Data is ready."  << std::endl;
    condVar.notify_one();
}
```

The function `setDataReady` performs the notification part of the synchronization - with the function `waitingForWork` as the waiting part of the synchronization.

Here is the same workflow with tasks.

```
// promiseFutureSynchronise.cpp

#include <future>
#include <iostream>
#include <utility>


void doTheWork(){
  std::cout << "Processing shared data." << std::endl;
}

void waitingForWork(std::future<void>&& fut){

    std::cout << "Worker: Waiting for work." << std::endl;
    fut.wait();
    doTheWork();
    std::cout << "Work done." << std::endl;

}

void setDataReady(std::promise<void>&& prom){
    std::cout << "Sender: Data is ready."  << std::endl;
    prom.set_value();


}

int main(){

  std::cout << std::endl;

  std::promise<void> sendReady;
  auto fut = sendReady.get_future();

  std::thread t1(waitingForWork, std::move(fut));
  std::thread t2(setDataReady, std::move(sendReady));

  t1.join();
  t2.join();

  std::cout << std::endl;

}
```

That was quite easy.

Thanks to `sendReady` (line 32) you get a future `fut` (line 34). The promise communicates using its return value `void` (`std::promise<void> sendReady`) that it is only capable of sending notifications. Both communication endpoints are moved into threads `t1` and `t2` (lines 35 and 36). The future waits using the call `fut.wait()` (line 15) for the notification of the promise: `prom.set_value()` (line 24).

The structure and the output of the program matches the corresponding program in the section condition variable.