Moving Threads

This lesson gives an overview of problems and challenges related to moving threads in C++.

Moving threads make the lifetime issues of threads even harder.

A thread supports the move semantic but not the copy semantic, the reason being the copy constructor of `std::thread` is set to `delete`: `thread(const thread&) = delete;`. Imagine what will happen if you copy a thread while the thread is holding a lock.

Let's move a thread.

```
// threadMoved.cpp

#include <iostream>
#include <thread>
#include <utility>

int main(){

  std::thread t([]{std::cout << std::this_thread::get_id();});
  std::thread t2([]{std::cout << std::this_thread::get_id();});

  t = std::move(t2);
  t.join();
  t2.join();

}
```

Both threads `t` and `t2` should do their simple job: printing their IDs. In addition to that, thread `t2` will be moved to `t` (line 12). At the end, the main thread takes care of its children and joins them. But wait, the result is very different from my expectations.

What is going wrong? We have two issues:

1. By moving the thread `t2` (taking ownership), `t` gets a new callable unit

and its destructor will be called. As a result, `t`'s destructor calls `std::terminate` because it is still joinable.

2. Thread t2 has no associated callable unit. The invocation of join on a thread without callable unit leads to the exception `std::system_error`.

Knowing this, fixing the errors is straightforward.

```
// threadMovedFixed.cpp

#include <iostream>
#include <thread>
#include <utility>

int main(){

  std::thread t([]{std::cout << std::this_thread::get_id() << std::endl;});
  std::thread t2([]{std::cout << std::this_thread::get_id() << std::endl;});

  t.join();
  t = std::move(t2);
  t.join();

  std::cout << "\n";
  std::cout << std::boolalpha << "t2.joinable(): " << t2.joinable() << std::endl;

}
```

The result is that thread `t2` is not joinable anymore.