Multithreading: Threads

This lesson illustrates the best practices used to implement multithreaded applications in C++.

WE'LL COVER THE FOLLOWING

Threads

Threads are the basic building blocks for writing concurrent programs.

Use tasks instead of threads

```
// asyncVersusThread.cpp

#include <future>
#include <thread>
#include <iostream>

int main(){

  std::cout << std::endl;

  int res;
  std::thread t([&]{ res = 2000 + 11; });
  t.join();
  std::cout << "res: " << res << std::endl;

  auto fut= std::async([]{ return 2000 + 11; });
  std::cout << "fut.get(): " << fut.get() << std::endl;

  std::cout << std::endl;

}
```

Based on the program, there are a lot of reasons for preferring tasks over threads. These are the main reasons:

- you can use a safe communication channel for returning the result of the communication. If you use a shared variable, you have to synchronize the access to it.
- you can quite easily return values, notifications, and exceptions to the caller.

With extended futures we get the possibility to compose futures and build highly sophisticated workflows. These workflows will be based on the continuation `then`, and the combinations `when_any` and `when_all`.

Be extremely careful if you detach a thread

The following code snippet requires our full attention.

```
#include <iostream>
#include <thread>

int main(){

  std::string s{"C++11"};
  std::thread t([&s]{ std::cout << s << std::endl; });
  t.detach();
}
```

Because thread `t` is detached from the lifetime of its creator, two race conditions can cause undefined behavior.

1. Thread `t` may outlive the lifetime of its creator. The consequence is that `t` refers to a non-existing `std::string`
2. The program shuts down before thread `t` can do its work because the lifetime of the output stream `std::cout` is bound to the lifetime of the main thread

Consider using an atomic joining thread

A thread `t` with a callable unit is called joinable if neither a `t.join()` nor a `t.detach()` call happened. The destructor of a joinable thread throws the `std::terminate` exception. In order not to forget the `t.join()`, you can create your own wrapper around `std::thread`. This wrapper checks in the constructor, if the given thread is still joinable, and joins the given thread in the destructor.

You don't have to build this wrapper on your own; Use the `scoped_thread` from Anthony Williams or the `gsl::joining_thread` from the guideline support library.

Now let's practice more about data sharing and multithreading.