Managing Thread Lifetime

This lesson gives an overview of how to use the join and detach functions to properly end thread execution in C++.

WE'LL COVER THE FOLLOWING

The parent has to take care of its children - a simple principle that has significant consequences for the lifetime of a thread. This small program starts a thread that displays its ID:

```
// threadWithoutJoin.cpp

#include <iostream>
#include <thread>

int main(){

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

}
```

But the program will not print the ID. What's the reason for this exception? Let's figure it out!

`join` & `detach` functions

The lifetime of a created thread `t` ends with its callable unit. Therefore, the creator has two choices:

1. It can wait until its child is done: `t.join()`.

2. It can detach itself from its child: `t.detach()`.

A `t.join()` call is useful when the subsequent code relies on the result of the calculation performed in the thread. `t.detach()` permits the thread to execute independently from the thread handle `t`; therefore, the detached thread will run for the lifetime of the executable. Typically, you use a detached thread for a long-running background service such as a server.

A thread `t` with a callable unit (you can create threads without 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; this was the reason the program execution of `threadWithoutJoin.cpp` terminated with an exception. If you invoke `t.join()` or `t.detach()` more than once on a thread `t`, you get a `std::system_error` exception.

Solution

The solution to this problem is quite simple: call `t.join()`

```
// threadWithJoin.cpp

#include <iostream>
#include <thread>

int main(){

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

  t.join();

}
```