Thread Lifetime Management: Warnings and Tips

Some caveats and tips on the lifetime of threads in C++ coming your way...

WE'LL COVER THE FOLLOWING

Warnings

The Challenge of `detach` : Of course you can use `t.detach()` instead of `t.join()` in the last program. The thread `t` is not joinable any more; therefore, its destructor didn't call `std::terminate` . But now you have another issue. The program behaviour is undefined because the main program may complete before the thread `t` has time to complete its workpackage; therefore, its lifetime is too short to display the id. For more details, see lifetime issues of variables.

Tips

`scoped_thread` by Anthony Williams

If it's too bothersome for you to manually take care of the lifetime of your threads, you can encapsulate a `std::thread` in your own wrapper class. This class should automatically call join in its destructor. Of course you can go the other way and call detach, but there is an issue with detach.

> Anthony Williams created such a useful class and presented it in his excellent book Concurrency in Action. He called the wrapper `scoped_thread`. `scoped_thread` gets a thread `t` in its constructor and checks if `t` is still joinable. If the thread `t` passed into the constructor is not joinable anymore, there is no need for the `scoped_thread`. If `t` is joinable, the constructor calls `t.join()`. Because the copy constructor and copy assignment operator are declared as `delete`, instances of `scoped_thread` can not be copied to or assigned from.

```
// scoped_thread.cpp

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


class scoped_thread{
std::thread t;
public:
 explicit scoped_thread(std::thread t_): t(std::move(t_)){
   if (!t.joinable()) throw std::logic_error("No thread");
 }
 ~scoped_thread(){
   t.join();
 }
 scoped_thread(scoped_thread&)= delete;
 scoped_thread& operator=(scoped_thread const &)= delete;
};
```