std::scoped_lock

Defined in header <mutex>

```
template< class... MutexTypes >
class scoped_lock;
```
(since C++17)

The class scoped_lock is a mutex wrapper that provides a convenient RAII-style mechanism for owning one or more mutexes for the duration of a scoped block.

When a scoped_lock object is created, it attempts to take ownership of the mutexes it is given. When control leaves the scope in which the scoped_lock object was created, the scoped_lock is destructed and the mutexes are released, in reverse order. If several mutexes are given, deadlock avoidance algorithm is used as if by std::lock.

The scoped_lock class is non-copyable.

Template parameters

MutexTypes - the types of the mutexes to lock. The types must meet the *Lockable* requirements unless `sizeof...(MutexTypes)==1`, in which case the only type must meet *BasicLockable*

Member types

Member type	Definition
mutex_type (if sizeof...(MutexTypes)==1)	Mutex, the sole type in MutexTypes...

Member functions

(constructor)	constructs a scoped_lock, optionally locking the given mutexes (public member function)
(destructor)	destructs the scoped_lock object, unlocks the underlying mutexes (public member function)
operator= [deleted]	not copy-assignable (public member function)

Example

The following example uses std::scoped_lock to lock pairs of mutexes without deadlock and is RAII-style.

Run this code

```
#include <mutex>
#include <thread>
#include <iostream>
#include <vector>
#include <functional>
#include <chrono>
#include <string>

struct Employee {
    Employee(std::string id) : id(id) {}
    std::string id;
    std::vector<std::string> lunch_partners;
    std::mutex m;
    std::string output() const
    {
        std::string ret = "Employee " + id + " has lunch partners: ";
        for( const auto& partner : lunch_partners )
            ret += partner + " ";
        return ret;
    }
};

void send_mail(Employee &, Employee &)
{
    // simulate a time-consuming messaging operation
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

void assign_lunch_partner(Employee &e1, Employee &e2)
{
```

```
    static std::mutex io_mutex;
    {
        std::lock_guard<std::mutex> lk(io_mutex);
        std::cout << e1.id << " and " << e2.id << " are waiting for locks" << std::endl;
    }

    {
        // use std::scoped_lock to acquire two locks without worrying about
        // other calls to assign_lunch_partner deadlocking us
        // and it also provides a convenient RAII-style mechanism

        std::scoped_lock lock(e1.m, e2.m);

        // Equivalent code 1 (using std::lock and std::lock_guard)
        // std::lock(e1.m, e2.m);
        // std::lock_guard<std::mutex> lk1(e1.m, std::adopt_lock);
        // std::lock_guard<std::mutex> lk2(e2.m, std::adopt_lock);

        // Equivalent code 2 (if unique_locks are needed, e.g. for condition variables)
        // std::unique_lock<std::mutex> lk1(e1.m, std::defer_lock);
        // std::unique_lock<std::mutex> lk2(e2.m, std::defer_lock);
        // std::lock(lk1, lk2);
        {
            std::lock_guard<std::mutex> lk(io_mutex);
            std::cout << e1.id << " and " << e2.id << " got locks" << std::endl;
        }
        e1.lunch_partners.push_back(e2.id);
        e2.lunch_partners.push_back(e1.id);
    }

    send_mail(e1, e2);
    send_mail(e2, e1);
}

int main()
{
    Employee alice("alice"), bob("bob"), christina("christina"), dave("dave");

    // assign in parallel threads because mailing users about lunch assignments
    // takes a long time
    std::vector<std::thread> threads;
    threads.emplace_back(assign_lunch_partner, std::ref(alice), std::ref(bob));
    threads.emplace_back(assign_lunch_partner, std::ref(christina), std::ref(bob));
    threads.emplace_back(assign_lunch_partner, std::ref(christina), std::ref(alice));
    threads.emplace_back(assign_lunch_partner, std::ref(dave), std::ref(bob));

    for (auto &thread : threads) thread.join();
    std::cout << alice.output() << '\n'  << bob.output() << '\n'
              << christina.output() << '\n' << dave.output() << '\n';
}
```

Possible output:

```
alice and bob are waiting for locks
alice and bob got locks
christina and bob are waiting for locks
christina and alice are waiting for locks
dave and bob are waiting for locks
dave and bob got locks
christina and alice got locks
christina and bob got locks
Employee alice has lunch partners: bob christina
Employee bob has lunch partners: alice dave christina
Employee christina has lunch partners: alice bob
Employee dave has lunch partners: bob
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
LWG 2981 (https://cplusplus.github.io/LWG/issue2981)	C++17	redundant deduction guide from `scoped_lock<MutexTypes...>` was provided	removed

See also

unique_lock (C++11)	implements movable mutex ownership wrapper (class template)
lock_guard (C++11)	implements a strictly scope-based mutex ownership wrapper (class template)