1 Locks

There are two main mutex wrapper classes in the thread support library: **lock_guard** and **unique_lock** (there is also share_lock, but we will not discussion this class). The purpose of these classes is to manage mutexes in the thread support library.

1.1 lock_guard:

lock_guard: The lock-guard class template immediately locks the mutex when constructed, and unlocks it when the lock guard goes out of scope (and destroyed). This is useful because it's no longer necessary to remember to unlock your mutexes! We demonstrate this with a safe increment function.

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4
5  using namespace std;
6
7  mutex g_mutex;
8  int g_num = 0;
9
10 //Increment a number "x" safely by "k" units.
11 void safe_increment(int& x, int k)
12 {
13    lock_guard<mutex> guard(g_mutex);
14    for (int i = 0; i < k; i++)
15       x++;
16 }
17
18
19 int main()
20 {
21    thread t1(safe_increment, ref(g_num), 10000);
22    thread t2(safe_increment, ref(g_num), 10000);
23
24    cout << "Number is now: " << g_num << "\n";
25
26    t1.join();
27    t2.join();
28
29    system("pause");
30 }
```

Commenting line 13 will lead to unexpected behavior for large k.

Alternative usage is to first lock the mutex, and then have the lock guard adopt the mutex. Adoption means that the lock guard does not attempt to lock the mutex (because it is already locked). This is useful because you may want to lock multiple mutexes at once. Locking them sequentially could lead to a deadlock!

Deadlock: In order to safely apply a transaction between bank account, you need to acquire a lock on *both* accounts. If you sequentially locked the accounts, this can cause problems. Say you want to transfer money from account A to B. Imagine that at the same time you acquired a lock on account A that another thread acquired the lock on B for a reverse transaction. Each thread would be blocked from executing until the other let go of their lock: Therefore the threads would become stuck for eternity!

The function std::lock() can be first used to simultaneously lock multiple mutexes without the possibility of deadlock. We could use this to implement a safe transaction function.

```
1  #include <iostream>
2  #include <thread>
3  #include <mutex>
4  #include <chrono>
5  using namespace std;
6
```

```
7  mutex g_mutex_a, g_mutex_b;
8  int acct_a = 1500000;
9  int acct_b = 2500000;
10
11 //Safe transaction of amount "amount" between "acct_a" to ""acct_b".
12 void safe_transaction(int& acct_a, int& acct_b, int amount)
13 {
14    lock(g_mutex_a, g_mutex_b);
15    lock_guard<mutex> guard_a(g_mutex_a, adopt_lock);
16    lock_guard<mutex> guard_b(g_mutex_b, adopt_lock);
17
18    for (int i = 0; i < amount; i++)
19    {
20       acct_a--;
21       acct_b++;
22    }
23
24 }
25
26 int main()
27 {
28    thread t1(safe_transaction, ref(acct_a), ref(acct_b), 300000);
29    thread t2(safe_transaction, ref(acct_b), ref(acct_a), 250000);
30
31    t1.join();
32    t2.join();
33
34    //Should be 1,450,000 and 2,550,000
35    cout << "Account a balance is: " << acct_a << "\n";
36    cout << "Account b balance is: " << acct_b << "\n";
37
38    system("pause");
39 }
```

If you comment out the lock guards and mutexes etc., transactions are not properly registered in the system!

However lock guard has limited functionality (though it is lightweight and quick). The underlying mutex must be locked upon construction (or already locked, if you are adopting), and can only be unlocked upon destruction. Another wrapper called unique lock, on the other hand, can be created without locking the mutex. Also the mutex can be unlocked without waiting until destruction (which means that other threads are only blocked as long as necessary). We describe unique locks in the next section.

1.2 unique_lock

unique_lock is similar to lock_guard, but with more flexibility:

- The mutex associated with a lock_guard must be locked upon the guard's construction (if it is unlocked, construction locks it. If it is locked already, you need to pass "std::adopt_lock"). However unique_lock doesn't need to be locked upon construction: It can be locked or unlocked at any time.

  Initializing the unique_lock (e.g. `unique_lock UL(mutex1)`) will lock the underlying mutex. To initialize without locking, simple type `unique_lock UL(mutex1,defer_lock)` and lock later via `lock(UL)`.

- Also the only way to unlock the underlying mutex is for the lock guard to go out of scope (and be destroyed). This is not the case for a unique_lock, which can be unlocked at any time (e.g `UL.unlock()`). However it still has the benefit of automatically unlocking upon destruction (if it is locked! If it is unlocked, it doesn't attempt to unlock it).

- Ownership can be transferred.

Here is an example of where lock_guard would not be able to do the job efficiently:

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>
using namespace std;

mutex g_mutex;

//Increments an integer x a total of k times. It take a 1-second gap between increments.
void timed_increment(int& x, int k)
{
   //Initializes unique_lock without locking.
   unique_lock<mutex> UL(g_mutex, defer_lock);

   for (int i = 0; i < k; i++)
   {
      //Safely increment x.
      UL.lock();
      x++;
      cout << "Number is now " << x << ".\n";
      UL.unlock();

      //Wait for 1 second
      this_thread::sleep_for(chrono::milliseconds(1000));
   }
}

int main()
{
   int x = 0;

   thread t1(timed_increment, ref(x), 5);
   thread t2(timed_increment, ref(x), 5);
   thread t3(timed_increment, ref(x), 5);

   t1.join();
   t2.join();
   t3.join();

   system("pause");
}
```

Here we are trying to safely increment an integer x a number of times, waiting 1 second in between increments. It is not necessary to lock x the entire time! Only when it is being updated. This program locks x only when it is being updated, and unlocks while it is waiting and does not need a lock on x. This would be impossible to do efficiently with `lock_guard`. Done with unique lock, main only takes 5 seconds. With lock guard, it would take 15 because 5 would need to be locked during the entire function `timed_increment`.