C++11 Restrictions

- Rules exist for what a compiler can do
  - As-if must respect thread visibility
- Optimizations must respect memory ordering constraints
- No new visible state changes
  - No extra stores
  - It isn’t that this stuff wasn’t undefined behavior before, it wasn’t even a meaningful question

Atoms

- Generic wrapper: `std::atomic<Type>`
  - Integral types, pointer types, all types
- Standard set of operations
  - Load, store, cast, CAS, exchange, is_lock_free
- Some types have more
  - Fetch and add, sub, and, or, xor
  - Inc, dec
- May be implemented with a SW mutex or by hardware instructions

Memory Order

- Default: `memory_order_seq_cst`
- Atomic operations have optional arguments to specify minimum necessary ordering
- Don’t change unless you have a good reason and lots of time for thinking and don’t mind random impossible to replicate bugs.

```cpp
#include <thread>
#include <atomic>
#include <cassert>

std::atomic<bool> x = {false};
std::atomic<bool> y = {false};
std::atomic<int> z = {0};

void write_x() {
    x.store(true, std::memory_order_seq_cst);
}

void write_y() {
    y.store(true, std::memory_order_seq_cst);
}

void read_x_then_y() {
    while (!x.load(std::memory_order_seq_cst)) ;
    if (y.load(std::memory_order_seq_cst)) { ++z; }
}

void read_y_then_x() {
    while (!y.load(std::memory_order_seq_cst)) ;
    if (x.load(std::memory_order_seq_cst)) { ++z; }
}

int main() {
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join();
    b.join();
    c.join();
    d.join();
    assert(z.load() != 0);  // will never happen
}
```
memory_order_release (acquire)

• Like seq_cst, but no total global order across different locations
  – Previous example could break
• Mutex have this constraint

relaxed

• No ordering or visibility constraints, only atomicity
• Useful for counters and such

There are other orderings

Don’t mess with them

A note about volatile

• volatile is NOT a multi-threaded concept
• Multi-threaded code should not use or depend on it
• volatile is for memory mapped IO
• Limited sequential semantics
C++11 Fences

- atomic_thread_fence(order)
- Limited utility given other synchronization
- E.g. Relaxed checks
  - Scan something with relaxed loads
  - If set, fence to ensure acquire semantics
    - Without an acquire, just because you saw with a relaxed load a value from an atomic on another thread doesn’t mean you saw the values written before by that thread unless you do an acquire

Locks/Mutexes

- Like posix, a variety
  - Mutex, timed_mutex, recursive_mutex, recursive_timed_mutex
  - C++14: shared_timed_mutex
- RAII (resource acquisition is initialization)
  - Lock_guard, unique_lock
  - C++14 shared_lock
- Algorithms
  - Try_lock, lock

std::mutex

- Standard single-entrant mutex
  - Cannot lock twice in one thread
- lock – blocking lock
- try_lock – returns true if lock acquired
- Unlock

std::timed_mutex

- Like a plain mutex, but with timeout support
- Adds:
  - try_lock_for(duration)
    - Tries to lock for (about) duration
    - May fail spuriously
  - try_lock_until(time)
    - Tries to lock until time
    - May fail spuriously

std::mutex m;
std::map<int> m;

void foo(int x) {
    l.lock();
    m.insert(x);
    l.unlock();
}

std::timed_mutex m;

void f() {
    auto now = std::chrono::steady_clock::now();
    if(m.try_lock_until(now + std::chrono::seconds(10))) {
        std::cout << "hello world\n";
        m.unlock();
    }
}
std::recursive_mutex

- Reentrant mutex
- Once lock acquired, lock can be repeatedly acquired
- Lock released after matching number of unlocks
- Same api as std::mutex

std::recursive_time_mutex

- A reentrant mutex with timeout support
- try_lock_for
- try_lock_until
- Same matching lock/unlock rules as recursive mutex

C++14: shared_timed_mutex

- Essentially RW lock
- Exclusive (W) api same as timed_mutex
- Adds:
  - lock_shared, try_lock_shared, unlock_shared, try_lock_shared_for, try_lock_shared_until

RAII

- std::lock_guard<type>
- Acquires a mutex in the constructor, release it in the destructor

std::mutex m;
std::lock_guard<std::mutex> lg(m);

condition_variable

- Point to (multi) point synchronization
- A (group of) thread can wait until signaled
- notify_one – wake up one waiter
- notify_all – wake up all waiters
- wait – block until notify

- Like posix, has an associated mutex

```cpp
std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false, processed = false;

void worker_thread() {
  // Wait until main() sends data
  std::lock_guard<std::mutex> lk(m);
  cv.wait(lk, [](bool){return ready;});

  // after the wait, we own the lock.
  std::cout << "Worker thread is processing data\n";
  std::string data += " after processing";

  // Send data back to main()
  processed = true;
  std::cout << "Worker thread signals data processing completed!\n";
  lk.unlock();
  cv.notify_one();
}

int main() {
  std::thread worker(worker_thread);
  data = "Example data"
  std::lock_guard<std::mutex> lk(m);
  cv.wait(lk, [](bool){return ready;});
  std::cout << "main() signals data ready for processing!\n";
  cv.notify_one();
}

// wait for the worker
  cv.notify_one();
  cv.notify_one();
  std::cout << "Worker join\n";
  worker.join();
}
```
thread

• Class encapsulates control of thread
• Construction: (function, args...)  
  – Creates a new thread which executes function with arguments args
  – This is the main way to start a thread
• join – wait for thread to exit
• hardware_concurrency (static) – number of concurrent threads supported
  – May return 0 or something useless

To the future

• Promise and futures – standard methods of asynchronous data-flow
• Essentially single element, one-shot queue