

C++'s concurrency memory model

Hans-J. Boehm



Reflects contributions from many other ISO WG21 participants, academic work by many authors, etc.

Overview

C++ concurrency memory model overview

Many/most C++ programmers should know this.

C++ atomic operations overview

Quick intro to C++ weakly ordered atomics, and their challenges.

Some C++ programmers should know this.

Sometimes critical for performance, but commonly needed only by some low-level code.



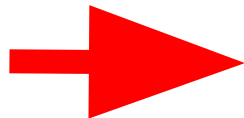
Background: Threads in C++

Ancient history (before C++11)

- C++ (and C) were single-threaded languages
 - All strictly conforming programs were sequential
- Made impractical by
 - Multicore processors
 - Many programs want to be structured as concurrent tasks.
- Many implementations supported some kind of add-on thread libraries, e.g.
 - Posix
 - Windows
- Problem: Threads impact compilation
 - Much confusion around detailed threads programming rules

Result: Broken code

```
while (...) {  
    if (threads) {  
        m.lock();  
    }  
    access/update g;  
    if (threads) {  
        m.unlock();  
    }  
}
```



```
r1 = g;  
while (...) {  
    if (threads) {  
        g = r1; m.lock(); r1 = g;  
    }  
    access/update r1;  
    if (threads) {  
        g = r1; m.unlock(); r1 = g;  
    }  
}  
g = r1;
```

- Arguably correct under Posix rules!
- I challenge you to program around it.
 - Remember that the code on the right may arise after transformations!

Confusion: Double-checked locking (Pugh et al.)

```
static bool init(false);

if (!init) {
    m.lock();
    if (!init) {
        initialize x;
        init = true;
    }
    m.unlock();
}
use x;
```

- Clever way to lazily initialize `x`.
- Advocated in text books.
- Broken.
 - Illegal in Posix.
 - Can fail on ARM.
 - Can fail with allowed compiler transformations.
 - Not fixable 10 yrs ago.

Recent history

- C++11
 - Added threads to the language.
 - Added many synchronization tools.
 - Clarified & fixed the rules.
- C++14 → C++20
 - Additional synchronization, `atomic_ref<>`
 - Various bug fixes & cleanups
- Ongoing
 - Interesting problems remain with detailed semantics of some `atomic<>` operations
 - ... in spite of widespread use

} I'll focus here

C++ 11 threads

```
void count_slowly(int n) {  
    for (int i = 0; i < n; ++i) {  
        cout << i << endl;  
        this_thread::sleep_for(...);  
    }  
}
```

...

```
thread my_thread = new thread(count_slowly, 10);  
count_slowly(10);  
my_thread.join();
```

Possible
output:

0
0
1
1
2
2
3
3
4
4
5
6
5
6
7
8
7
8
9
9

Also addressed “memory model”

Real threads share data.

How do shared variables work?

When I load the value of a variable that may have been assigned to by another thread, what value can it have?

Probably the most fundamental question about thread semantics.

Basic “sequentially consistent” threads programming model

Multiple instruction streams are executed one step at a time.

Machine repeatedly:

- Picks a thread at random
- Executes its next operation

Thread 1:

```
v = 1;  
w = 2;  
r1 = x;
```

Thread 2:

```
y = 3;  
r2 = z;
```

may be executed as

```
v = 1;  
y = 3;  
w = 2;  
r1 = x;  
r2 = z;
```

Reality is more complicated

What's an operation?

- Machine instructions aren't a programming language concept.
- And they're not indivisible anyway.

And compilers and hardware like to reorder instructions

- To make things go faster.

Simple model vs. Reality

x and y initially zero. (“Dekker’s” example.)

Thread 1:

```
x = 1;  
r1 = y;
```

Thread 2:

```
y = 1;  
r2 = x;
```

Simple sequentially consistent model:

- One thread goes first, $r1 = r2 = 0$ impossible.

Reality:

- Hardware and compiler reorders independent instructions,
 $r1 = r2 = 0$ is entirely possible.

Basic C++ (and C and Java) memory model

- Guarantee sequential consistency
- But restrict programs to make it:
 - Meaningful: Granularity doesn't matter.
 - Much easier: Reordering usually unobservable.



100% Guaranteed not to sink *

* May not be used in water

Restriction: No data races

Concurrent accesses to the same memory location are *data races*

- unless they are all reads/loads, or
- they are all synchronization or atomic operations.

Data races are disallowed.

- Officially they result in undefined behavior.

DRF guarantee(1)

No data races & no use of special “advanced” constructs

⇒ Sequential consistency

Dates back to Adve&Hill 90, Posix, Ada

DRF guarantee (2)

This has data races:

(*x* and *y* initially zero. “Dekker’s” example.)

Thread 1:

```
x = 1;  
r1 = y;
```

Thread 2:

```
y = 1;  
r2 = x;
```

DRF Guarantee (3)

This does not have data races:

Thread 1:

```
v = 1;  
w = 2;  
r1 = x;
```

Thread 2:

```
y = 3;  
r2 = z;
```

Sequential consistency is easy:

- Reordering is not observable.
- Implementation can “cheat” as much as it wants.

Important note:

Data races are defined with respect to sequential consistency. With everything initially false:

Thread 1:

```
if (x) y = true;
```

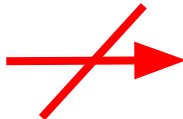
Thread 2:

```
if (y) x = true;
```

Does not have a data race!

Restricted guarantee is still important(1)

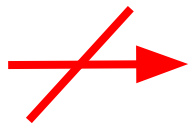
- Compiler cannot in general transform

```
if (x < 2) y = 17;  
if (x > 2) y = 17;    y_old = y;  
                                     y = 17;  
                                     if (x == 2) y = y_old;
```

- Multicore architectures must essentially have byte stores.

```
struct {char a; char b; } x;
```

```
x.a = 1;
```



```
x = <16 bits consisting of (1, x.b)>;
```

Restricted guarantee is still important(2)

- Questionable transformation at the beginning is clearly illegal.

Real code communicates via shared variables

Synchronization primitives to avoid data races:

- Locks or mutexes (`std::mutex`): prevent concurrent access.
- Atomic objects (`std::atomic<T>`): Allow concurrent access and force implementation to preserve sequentially consistent semantics in presence of concurrent access.

Both preserve interleaving / sequentially consistent semantics by default.

- So long as there are no data races.

Concurrent data-race-free increment

With a mutex:

```
int count;  
mutex m;
```

```
void incr_count()  
{  
    lock_guard<mutex> _(m);  
    ++count;  
}
```

With an atomic variable:


```
atomic<int> count;
```

```
void incr_count()  
{  
    count.fetch_add(1);  
    // or just ++count;  
}
```

Revisiting double-checked locking

```
if (!x_init) {  
    m.lock();  
    if (!x_init) {  
        initialize x;  
        x_init = true;  
    }  
    m.unlock();  
}  
use x;
```

```
if (!x_init) {  
    m.lock();  
    if (!x_init) {  
        initialize x;  
        x_init = true;  
    }  
    m.unlock();  
}  
use x;
```



Double-checked locking fixed

```
atomic<boolean> x_init(false);  
mutex x_init_mtx;
```

```
if (!x_init) {  
    lock_guard<mutex> _(x_init_mtx);  
    if (!x_init) {  
        initialize x;  
        x_init = true;  
    }  
}
```

Atomic objects vs. mutexes

- Mutexes are usually far simpler to use.
 - And should be the normal go-to mechanism.
- Clever algorithms using atomics may be faster.
 - Sometimes much faster.
 - Or not. It's complicated.
- Atomics are often a better match for signal/interrupt handlers, cross-process communication.
- Atomics are used surprisingly frequently.
- Atomics interact more closely with the memory model.

Dekker's example corrected, with atomics

At least one load returns 1:

```
atomic<int> x(0), y(0);
```

Thread 1:

```
x.store(1);  
r1 = y.load();
```

Thread 2:

```
y.store(1);  
r2 = x.load();
```

Basic memory model generally works well

- Concurrent programming is still hard, but
 - Programming rules are clear.
 - Compilation rules are clear.
 - Modern compilers generally follow them.
 - No extra stores.
 - Incompatible hardware is mostly dead.
 - Byte stores supported where it matters.
- Supported reasonably well since C++11
 - and Java since 2005, and arguably Ada before that.

But this isn't the whole story:

Preservation of sequential consistency is somewhat expensive / slow:

- Typically 5–60 extra cycles per load or store (near best case). (Loads are free on x86.)
- Very dependent on microarchitecture.



[Image by Pexels from Pixabay](#)

C++ (and C, Java, ...) provide an explicit escape:

- Weakly ordered atomic accesses.
E.g. `x.load(std::memory_order::acquire)`
- Waive DRF sequential consistency guarantees for speed.
- Greatly increased complexity.
- Widely used.
 - Sometimes correctly.



[Christopher Batt](#)

C++ weakly ordered accesses:

- `memory_order_acquire`, `memory_order_release`
- `memory_order_consume`
- `memory_order_relaxed`

(In C++ 20, `memory_order_yyy` can also be spelled `memory_order::yyy`)

(There are also memory fences. I won't talk about them. Think of them as even more esoteric.)

memory_order_acquire, memory_order_release

`x.store(a, memory_order_release)` makes memory effects visible to a `x.load(memory_order_acquire)` that returns *a*.

But `x.store(memory_order_release); y.load(memory_order_acquire)` can be reordered. Dekker's example doesn't work.

memory_order_acquire, memory_order_release

`int x` and `atomic<bool> y` initially zero/false. (“message passing” example.)

Thread 1:

```
x = 1;  
y.store(true, memory_order_release);
```

Thread 2:

```
if (y.load(memory_order_acquire))  
    assert(x == 1);
```

If the load sees true, then later memory accesses in thread 2 must see earlier accesses in thread 1.

In terms of the standard, if the load sees true:

- The store synchronizes with the load.
- `x = 1;` is sequenced before the store.
- The load is sequenced before the assert.
- Thus `x = 1;` happens before the assert.
- Since there is a happens-before relationship
 - There is no data race on `x`.
 - The assert must not see an earlier value of `x` than 1.

Double-checked locking, sped up a bit

```
atomic<boolean> x_init(false);
```

```
if (!x_init.load(memory_order_acquire) {  
    lock_guard<mutex> _(x_init_mtx);  
    if (!x_init.load(memory_order_acquire)) {  
        initialize x;  
        x_init.store(true, memory_order_release);  
    }  
}
```


Mutexes behave like acquire/release atomics

Mutex `unlock()` makes prior memory operations visible to the next `lock()` on that mutex.

Mutex `unlock()` synchronizes with next `lock()` on that mutex.

At the expense of interesting hazards

Subtle interaction with locks:

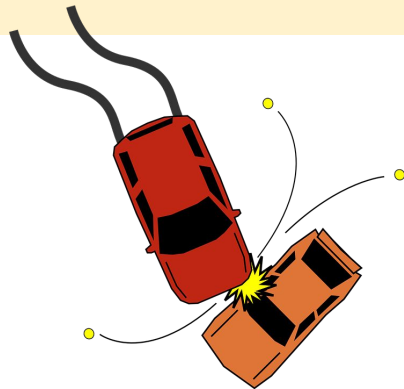
Thread 1:

```
x.store(true, memory_order_release);  
{ lock_guard<mutex> _(m1); }  
if (!y.load(memory_order_acquire))  
    turn_EW_lights_green();
```

Thread 2:

```
y.store(true, memory_order_release);  
{ lock_guard<mutex> _(m2); }  
if (!x.load(memory_order_acquire))  
    turn_NS_lights_green();
```

Critical sections do not necessarily order accesses.
Memory model allows movement *into* critical sections.
(Does work if both both mutexes are the same.)



memory_order_acquire, memory_order_release

- Trickier to reason about* than default `memory_order_seq_cst`.
- But not fundamentally problematic.
- Commonly used for lazy initialization, passing data from one thread to another, ...

*Even that's slightly controversial.

Brief note on `memory_order_consume`

Tries to address common & important `memory_order_acquire` special case:

```
Foo f; atomic Foo* x(nullptr);
```

Thread 1:

```
f.a = 17;  
x.store(&f, memory_order_release);
```

Thread 2:

```
r1 = x.load(<mo?>);  
if (r1 != nullptr)  
    r2 = r1->a; // initialized?
```

`memory_order_acquire` works for `<mo?>`, but

- Generates appreciably slower code on ARM, Power, ...

Unfortunately current `memory_order_consume` doesn't work.

We got it wrong. Please ignore.

memory_order_relaxed

C++ (or C) `memory_order_relaxed`:

- allows concurrent accesses to the same variable,
- allows independent accesses to different variables to be seen out of order by other threads.
- Even stores may be performed first in

```
r1 = x.load(memory_order_relaxed);  
y.store(1, memory_order_relaxed);
```

Many uses

Some benign ones:

- Non-concurrent access to an `atomic<T>`.
- For the initial load preceding a compare-and-swap implementing an atomic $x = f(x)$ update.
- Simple counters, read for debugging or after joining threads.

Result of “racing loads” is not critical for correctness.

Benign memory_order_relaxed use

Lazy initialization via double-checked locking

```
atomic<boolean> x_init(false);
```

```
if (!x_init.load(memory_order_acquire)) {  
    lock_guard<mutex> _(x_init_mtx);  
    if (!x_init.load(memory_order_relaxed)) {  
        initialize x;  
        x_init.store(true, memory_order_release);  
    }  
}
```

And many challenging ones:

- Owner field for a reentrant mutex.
- Lazy idempotent initialization.
 - Java-style identity hashCode access.
- Iterative numerical algorithms that don't care much whether they see the current or the last value.
- Reference count increments.

Believed to be correct, but not provably so according to spec.

The problem

- Very hard to specify correctly.
 - Again “dependencies” are the issue.
- Very hard to reason about.
 - The core problem appears to be theoretical.
 - But weird results are possible in practice.
- Widely used.
- Simple fixes cost performance.

My current scariest example:

```
// Foo has virtual f()  
atomic<Foo*> x(nullptr), y(nullptr);
```

Thread 1:

```
y.store(x.load(mo_rlx),  
        mo_rlx);
```

Thread 2:

```
Foo* r2 = y.load(mo_rlx);  
if (r2 == nullptr) {  
    r2 = new Foo();  
}  
x.store(r2, mo_rlx);  
r2->f(); // UB, since the Foo may not  
         // have actually been  
         // constructed (and never will be)!
```

See wg21.link/p1217 for more details.

In conclusion ...

No data races!

Either

- Stick to sequentially consistent atomics, or
- Be very careful you understand the rules!

Don't use `memory_order_consume`.

Avoid `memory_order_relaxed` unless either:

- There's no concurrent store, or
- The value returned by `load()` doesn't affect correctness.

But prefer `memory_order_relaxed` to data races!

Questions?

Backup slides

memory_order_consume(2)

Informally `<mo?>` must order loads to `r1` and `r2`:

Thread 1:

```
f.a = 17;  
x.store(&f, memory_order_release);
```

Thread 2:

```
r1 = x.load(<mo?>);  
if (r1 != nullptr)  
    r2 = r1 -> a; // initialized?
```

- Load to `r2` depends on address from load to `r1`.
- Modern CPUs enforce ordering implicitly in that case.
- At the hardware level, an ordinary load will do.

memory_order_consume(3)

So does `memory_order_relaxed` work?

Thread 1:

```
f.a = 17;  
x.store(&f, memory_order_release);
```

Thread 2:

```
r1 = x.load(memory_order_relaxed);  
if (r1 != nullptr)  
    r2 = r1 -> a; // initialized?
```

- Assuming naive compilation, yes!
- Assuming a real compiler, maybe ...
- Based on the language spec, no!

memory_order_consume(4)

What can go wrong?

- `x` is either null, or points to `f`. Compiler figures that out.
- Load into `r2` can be optimized to not wait for initial load:

Thread 1:

```
f.a = 17;  
x.store(&f, memory_order_release);
```

Thread 2:

```
r1 = x.load(memory_order_relaxed);  
if (r1 != nullptr)  
    r2 = f.a; // may be uninitialized!
```

- Hardware speculates branch condition is true.
- Load to `r2` completes before non-null value is read.

memory_order_consume(5)

The core problem:

For `memory_order_relaxed` to work in cases like this:

- Dependencies must be preserved.

To reduce execution time:

- Compilers break as many dependencies as possible.

memory_order_consume(6)

Recognized as critical from the beginning.

- In C++11, with a “clever”, though buggy, definition.
- Too difficult to implement properly.
 - Affects optimization of code unrelated to atomics.
- Too difficult to use effectively.
 - Too easy to specify unintended optimization constraints.

memory_order_relaxed

C++ (or C) `memory_order_relaxed`:

- allows concurrent accesses to the same variable,
- allows independent accesses to be seen out of order by other threads.
- Stores may be performed first in

```
r1 =rlx x;  
y =rlx 1;
```

Treat this as specification problem (for now)

- Next 2 examples are *not* believed to occur in practice.
- They're very hard to formally distinguish from actual reordering.
- Explanation as to how they might occur are necessarily a stretch.

Not “out-of-thin-air” result

```
atomic<int> x(0), y(0);
```

Thread 1:

```
// y = x;
```

```
int r1 =r1x x;
```

```
y =r1x r1;
```

Thread 2:

```
// x = y;
```

```
int r2 =r1x y;
```

```
x =r1x 42;
```

$r1 = r2 = x = y = 42$ is fine!

“out-of-thin-air” results (1)

```
atomic<int> x(0), y(0);
```

Thread 1:

```
// y = x;
```

```
int r1 =r1x x;
```

```
y =r1x r1;
```

Thread 2:

```
// x = y;
```

```
int r2 =r1x y;
```

```
x =r1x r2;
```

r1 = r2 = x = y = 42 should be disallowed!

“out-of-thin-air” results (1)

```
atomic<int> x(0), y(0);
```

Thread 1:

```
// y = x;  
  
int r1 =r1x x; // guess 42  
  
y =r1x r1;  
  
// Confirm speculation
```

Thread 2:

```
// x = y;  
  
int r2 =r1x y; // guess 42  
  
x =r1x r2;  
  
// Confirm speculation
```

Formally each load observes the other thread's store, as before.

“out-of-thin-air” results (2)

```
atomic<unsigned int> x(0), y(0);
```

Thread 1:

```
int r1 =r1x x;  
f(r1);
```

Thread 2:

```
int r2 =r1x y;  
g(r2);
```

where **f** and **g** are library routines, with precondition: argument ≤ 2 .

“out-of-thin-air” results (2)

```
atomic<unsigned int> x(0), y(0);
```

Thread 1:

```
int r1 =r1x x; // Guesses 42  
f(r1); // Out-of-bounds access  
        // writes 42 to y.
```

Thread 2:

```
int r2 =r1x y; // Guesses 42  
g(r2); // Out-of-bounds access  
        // writes 42 to x.
```

where **f** and **g** are library routines, with precondition: argument ≤ 2 .

“out-of-thin-air” results (3)

- Long-standing open problem:
 - Correctly define and prohibit out-of-thin-air results.
 - Without prohibiting necessary reordering
- Java, C11, C++11 tried hard and failed.

Out-of-thin-air in C++14/17 standard, C++20 draft

Defined only by example!

“Implementations should ensure that no **“out-of-thin-air”** values are computed that circularly depend on their own computation.”

“out-of-thin-air” results (4)

- Were widely believed to not occur in practice.
- Prevent precise reasoning about code & compilers.
 - E.g. can't prove that code is not exploitable.

Not as theoretical as we thought ...

Current implementations do allow out-of-thin-air-like results!

- Unclear if it happens in production code, but
- If we want precise reasoning, accurate tools:
 - We need to change implementations!
 - Well maybe. Controversial.

Very similar examples can occur with real compilers!

```
atomic<int> x(0), y(0);
```

Thread 1:

```
y =rlx x;
```

Thread 2:

```
bool assigned_42 = false;  
r2 =rlx y;  
if (r2 != 42) {  
    assigned_42 = true;  
    r2 = 42;  
}  
x =rlx r2;
```

Can result in

$x = y = 42$ and

$assigned_42 = false!$

Transformation (1)

```
atomic<int> x(0), y(0);
```

Thread 1:

```
y =rlx x;
```

Thread 2:

```
bool assigned_42 = false;  
r2 =rlx y;  
if (r2 != 42) {  
    assigned_42 = true;  
    r2 = 42;  
}  
x =rlx r2; // r2 always = 42
```


Transformation (2)

```
atomic<int> x(0), y(0);
```

Thread 1:

```
y =rlx x;
```

Thread 2:

```
bool assigned_42 = false;  
r2 =rlx y;  
if (r2 != 42) {  
    assigned_42 = true;  
    r2 = 42; // dead  
}  
x =rlx 42;
```

Transformation (3)

```
atomic<int> x(0), y(0);
```

Thread 1:

```
y =rlx x;
```

Thread 2:

```
bool assigned_42 = false;  
r2 =rlx y;  
if (r2 != 42) {  
    assigned_42 = true;  
  
}  
x =rlx 42;
```

Transformation (4)

```
atomic<int> x(0), y(0);
```

Thread 1:

```
y =rlx x;
```

Thread 2:

```
bool assigned_42;  
r2 =rlx y;  
assigned_42 = (r2 != 42);
```

```
x =rlx 42;
```

Transformation (5)

```
atomic<int> x(0), y(0);
```

Thread 1:

```
y =rlx x;
```

Thread 2:

```
bool assigned_42;  
r2 =rlx y;  
assigned_42 = (r2 != 42);  
  
x =rlx 42; // Can be advanced
```

Transformation (6)

```
atomic<int> x(0), y(0);
```

Thread 1:

```
y =rlx x;
```

The crucial
transformation!
On ARM, Nvidia, hardware
can do this!

Thread 2:

```
bool assigned_42;  
x =rlx 42;  
// Thread 1 executes here.  
r2 =rlx y;  
assigned_42 = (r2 != 42);
```

How?

- There is no prior allocation.
- New expression is evaluated at most once.
- Decompose the new expression.

Thread 2:

```
Foo* r2 =r1x y;  
if (r2 == nullptr) {  
    q = first loc in malloc heap;  
    update malloc heap;  
    r2 = new(q) Foo();  
}  
x =r1x r2;  
r2->f();
```

How? (2)

- **q** computation doesn't depend on anything. (It may be constant).
- Move q computation up.
- **r2** is always null or **q**.

Thread 2:

```
q = first loc in malloc heap;  
Foo* r2 =r1x y;  
if (r2 != q) {  
    update malloc heap;  
    r2 = new(q) Foo();  
}  
x =r1x r2;  
r2->f();
```

How? (3)

- **r2** is always **q** after conditional.
- **x** is always assigned **q**.

Thread 2:

```
q = first loc in malloc heap;  
Foo* r2 =r1x y;  
if (r2 != q) {  
    update malloc heap;  
    r2 = new(q) Foo();  
}  
x =r1x q;  
q->f();
```


How? (4)

- Assignment to **x** can be moved up.
- Thread 1 runs after assignment to **x**.
- **Foo** constructor is not executed.
- Call to **q->f** takes wild branch.

Thread 2:

```
q = first loc in malloc heap;  
x =r1x q;  
Foo* r2 =r1x y;  
if (r2 != q) {  
    update malloc heap;  
    r2 = new(q) Foo();  
}  
q->f();
```

A few other recent proposals

Define `memory_order_relaxed` as implemented:

- Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer, “A promising semantics for relaxed memory concurrency”, POPL 2017.
- Chakraborty, Vafeiades, “Grounding Thin-Air Reads with Event Structures”, POPL 2019.
- Batty, Owens, Paradis, Paviotti, and Wright, Modular Relaxed Dependencies: A new approach to the Out-Of-Thin-Air Problem (Fix current spec without disallowing behavior.) <http://wg21.link/p1780>. (Latest version only on committee site.)

An interesting alternative approach:

- Matthew D. Sinclair, Johnathan Alsop, Sarita V. Adve: “Chasing Away RATs: Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems”. ISCA 2017: 161-174