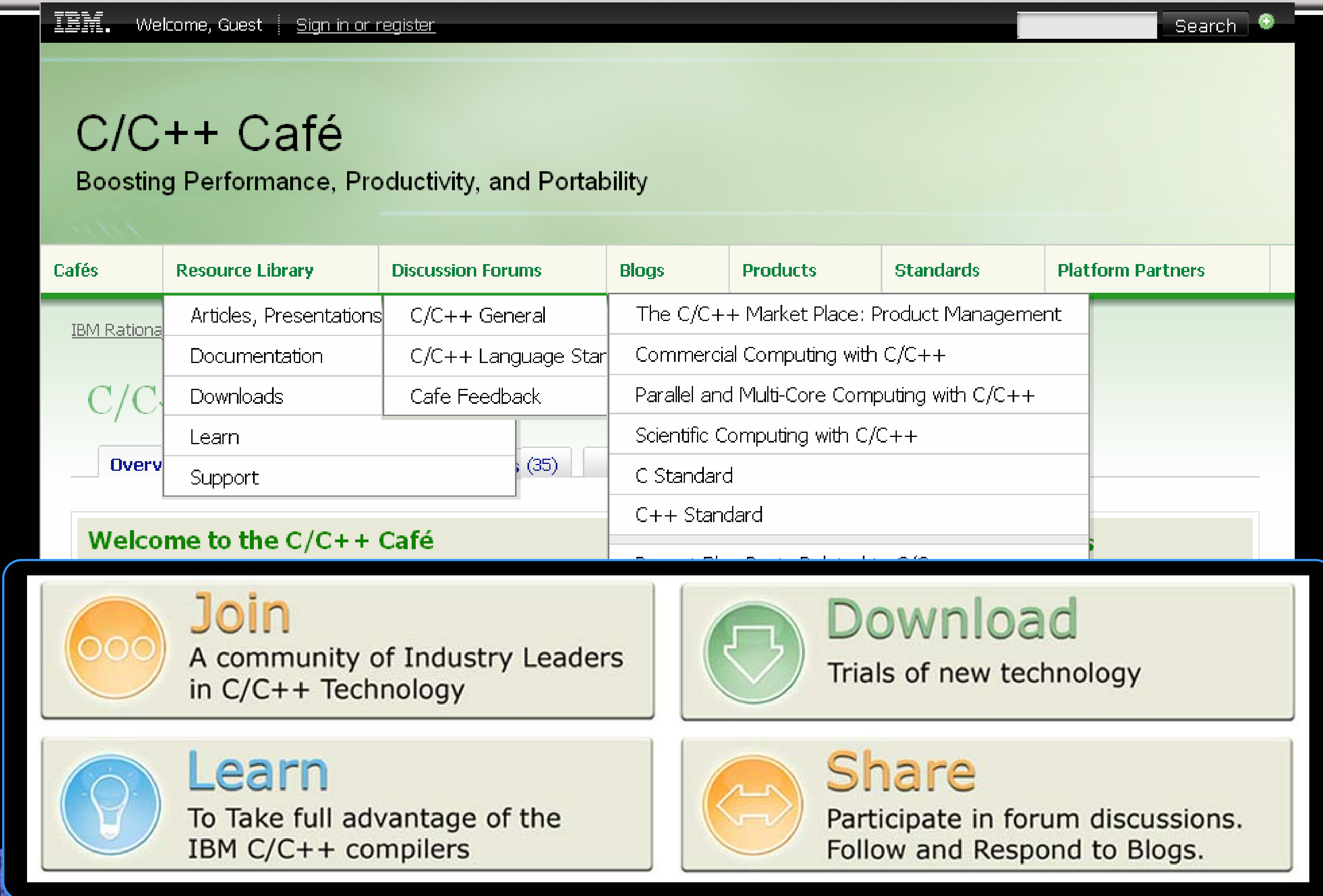
Multithreaded C++0x: the Dawn of a new Standard

Michael Wong
michaelw@ca.ibm.com
IBM Toronto Lab
Canadian C++ Standard Committee

IBM Rational Disclaimer

The IBM Rational C/C++ Café

ibm.com/rational/cafe/community/ccpp



Agenda

- **Concurrent C++0x examples**
- Atomics header
- Atomic types
- Atomic operations
- Atomic relations
- Atomic ordering
- Q/A

Hello Concurrent World

```
#include <iostream>
#include <thread> //#1
void hello() //#2
{
   std::cout<<"Hello Concurrent World"<<std::endl;
}
int main()
{
   std::thread t(hello); //#3
   t.join(); //#4
}
```

Is this valid C++ today? Are these equivalent?

```
int x = 0;

atomic<int> y = 0;
```

Thread 1:

```
  x = 17;
  y.store(1,
  memory_order_release);
  // or:       y.store(1);
```

Thread 2:

```
  while
  (y.load(memory_order_acquir
  e) != 1)
  // or:       while
  (y.load() != 1)

  assert(x == 17);
```

```
int x = 0;

atomic<int> y = 0;
```

Thread 1:

```
  x = 17;
  y = 1;
```

Thread 2:

```
  while (y != 1)
         continue;
  assert(x == 17);
```

Memory Model

- Locks and atomic operations communicate non-atomic writes between two threads
- Volatile is not atomics
- Memory races cause undefined behavior
- Some optimizations are no longer legal
- Compiler may assume some loops terminate

Message shared memory

- **Writes are explicitly communicated**
  - Between pairs of threads
  - Through a lock or an atomic variable
- **The mechanism is acquire and release**
  - One thread releases its memory writes
    - V=32; atomic_store_explicit(&a,3, memory_order_release);
  - Another thread acquires those writes
    - i=atomic_load_explicit(&a, memory_order_acquire); i+v;

What is a memory location

- **A non-bitfield primitive data object**
- **A sequence of adjacent bitfields**
  - Not separated by a structure boundary
  - Not interrupted by the null bitfield
  - Avoid expensive atomic read-modify-write operations on bitfields

Data race condition

- A non-atomic write to a memory location in one thread
- A non-atomic read from or write to that same location in another thread
- With no happens-before relations between them
- Is undefined behaviour

Effect on compiler optimization

- **Some rare optimizations are restricted**
  - Fewer speculative writes
  - Fewer speculative reads
- **Some common optimizations can be augmented**
  - They may assume that loops terminate
  - Nearly always true

Atomics: To Volatile or Not Volatile

- **Too much history in volatile to change its meaning**
- **It is not used to indicate atomicity like Java**
- **Volatile atomic means something from the environment may also change this in addition to another thread**

Requirements on atomics

- **Static initialization**
- **Reasonable implementation on current hardware**
- **Relative novices can write working code**
- **Experts can performance efficient code**

Consistency problem

- **X and y are atomic and initially 0**
  - Thread 1: x=1;
  - Thread 2: y=1;
  - Thread 3: if (x==1 && y==0)
  - Thread 4: if (x==0 && y==1)
- **Are both conditions exclusive?**
  - Is there a total store order?
- **The hardware/software system may not provide it**
- **Programming is harder without it**

Consistency models

- **Sequentially consistent**
  - What is observed is consistent with a sequential ordering of all events in the system
    - But comes with a very heavy cost
- **Weaker models**
  - More complex to code for some
    - But very efficient
- **What we decided**
  - Default is sequential consistency
  - But allow weaker semantics explicitly

Atomic Library (N2427)

- The problem:
  - **Would like to implement, for example, counters, without locks using atomics**

```
atomic<int> x;                    int x;
void increment() {                mutex m;
    ++x;                          void increment(){
                                          lock_guard_(m);
                                          x=x+1;
}                                 }
```

- **Advantages**
  - Sometimes enables much better performance
  - No space for locks, sometimes simpler.
  - Potentially safe for use with signal handlers, across processes.

Atomic DCL

```
T x;
atomic_bool x_init(false);
mutex m;
if (!x_init) {
    lock_guard _(m);
    if (!x_init) {
        x = ….
        x_init = true;
    }
}
use x;
```

- Note: Atomics are still tricky. Only a single memory operation at a time is atomic!

Atomic Design

- Want shared variables
  - **that can be concurrently updated without introducing data race,**
  - **that are atomically updated and read**
    - **half updated states are not visible,**
- that are implemented without lock overhead whenever the hardware allows,
- that provide access to hardware atomic read-modify write (fetch-and-add, xchg, cmpxchg) instructions whenever possible.

Race Free semantics and Atomic Memory operations

- **If a program has a race, it has undefined behavior**
  - This is sometimes known as “catch fire” semantics
  - No compiler transformation is allowed to introduce a race
    - no invented writes
    - Possibly fewer speculative stores **and (potentially) loads**
- **There are atomic memory operations that don’t cause races**
  - Can be used to implement locks/mutexes
  - Also useful for lock-free algorithms
- **Atomic memory operations are expressed as library function calls**
  - Reduces need for new language syntax

Atomic Operations and Type

- **Data race: if there is no enforced ordering between two accesses to a single memory location from separate threads, one or both of those accesses is not atomic, and one or both is a write, this is a data race, and causes undefined behavior.**
- **These types avoid undefined behavior and provide an ordering of operations between threads**

Standard Atomic Types

- **#include <cstdatomic>**
- **atomic_flag**
- **atomic_bool**
- **atomic_address**
- **Integral types:**
  - atomic_char, atomic_schar, atomic_uchar, atomic_short, atomic_ushort, atomic_int, atomic_uint, atomic_long, atomic_ulong, atomic_llong, atomic_ullong atomic_char16_t, atomic_char32_t, atomic_wchar_t
- **Typedefs like those in <cstdint>**
  - atomic_int_least8_t, atomic_uint_least8_t, atomic_int_least16_t, atomic_uint_least16_t, atomic_int_least32_t, atomic_uint_least32_t, atomic_int_least64_t, atomic_uint_least64_t, atomic_int_fast8_t, atomic_uint_fast8_t, atomic_int_fast16_t, atomic_uint_fast16_t, atomic_int_fast32_t, atomic_uint_fast32_t, atomic_int_fast64_t, atomic_uint_fast64_t, atomic_intptr_t, atomic_uintptr_t, atomic_size_t, atomic_ssize_t, atomic_ptrdiff_t, atomic_intmax_t, atomic_uintmax_t
- **is_lock_free();**
- **Non-copyable, non-assignable**

Minimal atomics

- **Need 1 primitive data types that is a must, m**ost modern hardware has instructions to implement the atomic operations
  - **for small types**
  - **and bit-wise comparison, assignment (which we require)**
  - atomic_flag type

    static std::atomic_flag v1= ATOMIC_FLAG_INIT
    If (atomic_flag_test_and_set(&v1))
        atomic_flag_clear(&v1);
- **For other types, hardware, atomic operations may be emulated with locks.**
  - Sometimes this isn.t good enough:
    - across processes, in signal/interrupt handlers.
  - is_lock_free() returns false if locks are used, and operations may block.
- **Operations on variable have attributes, which can be explicit**
  - Acquire=get other memory writes
  - Release=give my memory writes
  - Acq_and_rel=Acquire and release at the same time
  - Relaxed=no acquire or released, non-deterministic, not synchronizing with the rest of memory, but still sequential view of that variable
  - Seq-cst=Fully ordered,extra ordering semantics beyond acquire and releases, this is sequentially consistent
  - Consumed=dependecy-based ordering

Std::atomic_bool

- **Most basic std::atomic_bool, can be built from a non-atomic bool**
- **Can be constructed, initiialized, assigned from a plain bool**
- **assignment operator from a non-atomic bool does not return a reference to the object assigned to, but it returns a bool with the value assigned (like all other atomic types).**
  - prevents code that depended on the result of the assignment to have to explicitly load the value, potentially getting a modified result from another thread.
- **replace the stored value with a new one and retrieve the original value**
- **a plain non-modifying query of the value with an implicit conversion to plain bool**
- **RMW operation that stores a new value if the current value is equal to an expected value is compare_exchange_{weak/strong}();**
- **If we have spurious failure:**

```
bool expected=false; extern atomic_bool b; // set somewhere else
while(!b.compare_exchange_weak(expected,true) && !expected);
```

- **May not be lock free, need to check per instance**

Std::atomic_address

- similar operations as std::atomic_bool, can be constructed from non-atomic void*
- all operations take and return void*
- adds operations fetch_add() and fetch_sub() and += and -= as wrappers

Basic atomics

- **atomic<bool>**
  - Load, store, swap, cas
- **atomic<int>**
  - Load, store, swap, cas
  - Fetch-and-(add, sub, and, or, xor)
- **atomic<void *>**
  - Load, store, swap, cas
  - Fetch-and-(add, sub)

Std::atomic_i ntegral

- this adds fetch_and, fetch_or, fetch_xor, and compound assignments like:
- +=,-=,&=,^=, pre and post increment and decrement
- missing division, multipication and shift operations, but atomic integrals are usually used as counters or bit masks, this is not a big loss
- all semantics match fetch_add and fetch_sub for atomic_address: returns old value
- the compound assignments return new value
- ++x increments the variable and returns new value, x++ increments the variable and returns old value
- result is the value of the associated integral type

Other Atomic facilities

- Specializations for integral types, pointers
  - **Provide atomic increment, decrement (++, --, +=, -=)**
  - **Note: x++ is very different from x = x + 1 !**
    - **Unlike Java volatiles, where both are probably wrong!**
- Non-template (C-like) atomic types
  - **Template specializations inherit from these**
- C-like stand-alone (atomic_) function interfaces.

Std::atomic <> template

- **std::atomic<> to create an atomic user-defined type**
- **Specializations for integral types derived from std::atomic_integral_type, and pointer types**
- **Main benefit of the template is atomic variants of user-defined types, can't be just any UDT, it must fit this criteria:**
  - must have trivial copy-assignment operator: no virtual functions or virtual bases and must use the compiler-generated copt-assignment operator
  - every base class and non-static data member of UDT must also have a trivial copy-assignment operator
  - Must be bitwise equality comparable
- **Only have**
  - load(), store()
- **Assignment and conversion to the UDT**
  - exchange(), compare_exchange_weak(), compare_exchange_strong()
  - assignment from and conversion to an instance of type T

Free functions

- **Designed to be C compatible, so they use pointers and not references**
- **overloaded for each atomic type**
- **all take a pointer to the atomic object as first parameter**
- **2 varieties**
  - one without the memory order tag
  - one with an _explict suffix and additional memory ordering tag, or tags
- **std::atomic_is_lock_free() comes in only one variety**
  - std::atomic_is_lock_free(&a) returns the same value as a.is_lock_free()
- **std::atomic_load(&a)returns the same value as a.load();**
- **std::atomic_load_explict(&a,std::memory_order_acquire) is the same as a.load(std::memory_order_acquire)**

Sequencing redefined for serial program

- **Sequence points are … gone!**
- **Sequence are now defined by ordering relations**
  - Sequence-before
  - Indeterminately-sequenced
- **A write/write or read/write pair relations**
  - That are not sequenced before
  - That are not indeterminately-sequenced
  - Results in undefined behaviour

Sequencing extended for parallel programs

- **Sequenced-before**
  - Provides intra-thread ordering
- **Synchronizes with (Acquire and release)**
  - Provide inter-thread ordering
- **Happens-before relation**
  - Between memory operations in different threads

Sequenced before

- **If a memory update or side-effect *a is-sequenced-before* another memory operation or side-effect *b*,**
  - then informally *a* must appear to be completely evaluated before *b* in the sequential execution of a single thread, e.g. all accesses and side effects of *a* must occur before those of *b*.
  - We will say that a subexpression *A* of the source program *is-sequenced-before* another subexpression *B* of the same source program to indicate that all side-effects and memory operations performed by an execution of *A* occur-before those performed by the corresponding execution of *B*, i.e. as part of the same execution of the smallest expression that includes them both.
- **We propose roughly that wherever the current standard states that there is a sequence point between *A* and *B*, we instead state that *A* is-sequenced-before *B*. This will constitute the precise definition of *is-sequenced-before* on subexpressions, and hence on memory actions and side effects.**

Cases

- **Function calls:**
  - The evaluations of the postfix expression and of the argument expressions are all unsequenced relative to one another. All side effects of argument expression evaluations are sequenced before the function is entered
- **Increment & Decrement:**
  - The value computation of the ++ expression is sequenced before the modification of the operand object.
- **Logical AND operator**
  - If the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression.
- **Conditional Operator**
  - Every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second or third expression.
- **Comma operator**
  - Every value computation and side effect associated with the left expression is sequenced before every value computation and side effect associated with the right expression.

Synchronizes with

- only between operations on atomic types
- operations on a data structure (locking a mutex) might provide this relationship if the data structire contains atomic types, and the operations on that data structure perform the appropriate operations internally
- definition:
  - **a suitably-tagged atomic write operation on a variable x synchronizes-with a suitably-tagged atomic read operation on x that reads the value stored by (a) that write, (b) a subsequent atomic write operation on x by the same thread that performed the initial write, or (c) an atomic read-modify-write operation on x (such as fetch_add() or compare_exchange_weak()) by any thread, that read the value written.**
- Store-release synchronizes-with a load-acquire

Happens before

- It specifies which operations see the effects of which other operations.
- An evaluation A happens before an evaluation B if:
  - **A is sequenced before B, or**
  - **A synchronizes with B, or**
  - **for some evaluation X, A happens before X and X happens before B.**

Happens-before

Thread 1

```
if (!x_init.ld_acq())
{
  lock();
  if (!x_init.ld…())
    x = …;
  x_init.store_rel(1);
  unlock();
}
… x …
```

#3 (x = …;)

#4 (x_init.store_rel(1);)

Thread 2

```
if (!x_init.ld_acq())
{
  lock();
  if (!x_init.ld…())

    x = …;
  x_init.store_rel(1);
  unlock();
}
… x …
```

#1

#2

Memory Ordering Operations

typedef enum memory_order {

memory_order_relaxed, memory_order_consume, memory_order_acquire,

memory_order_release, memory_order_acq_rel, memory_order_seq_cst

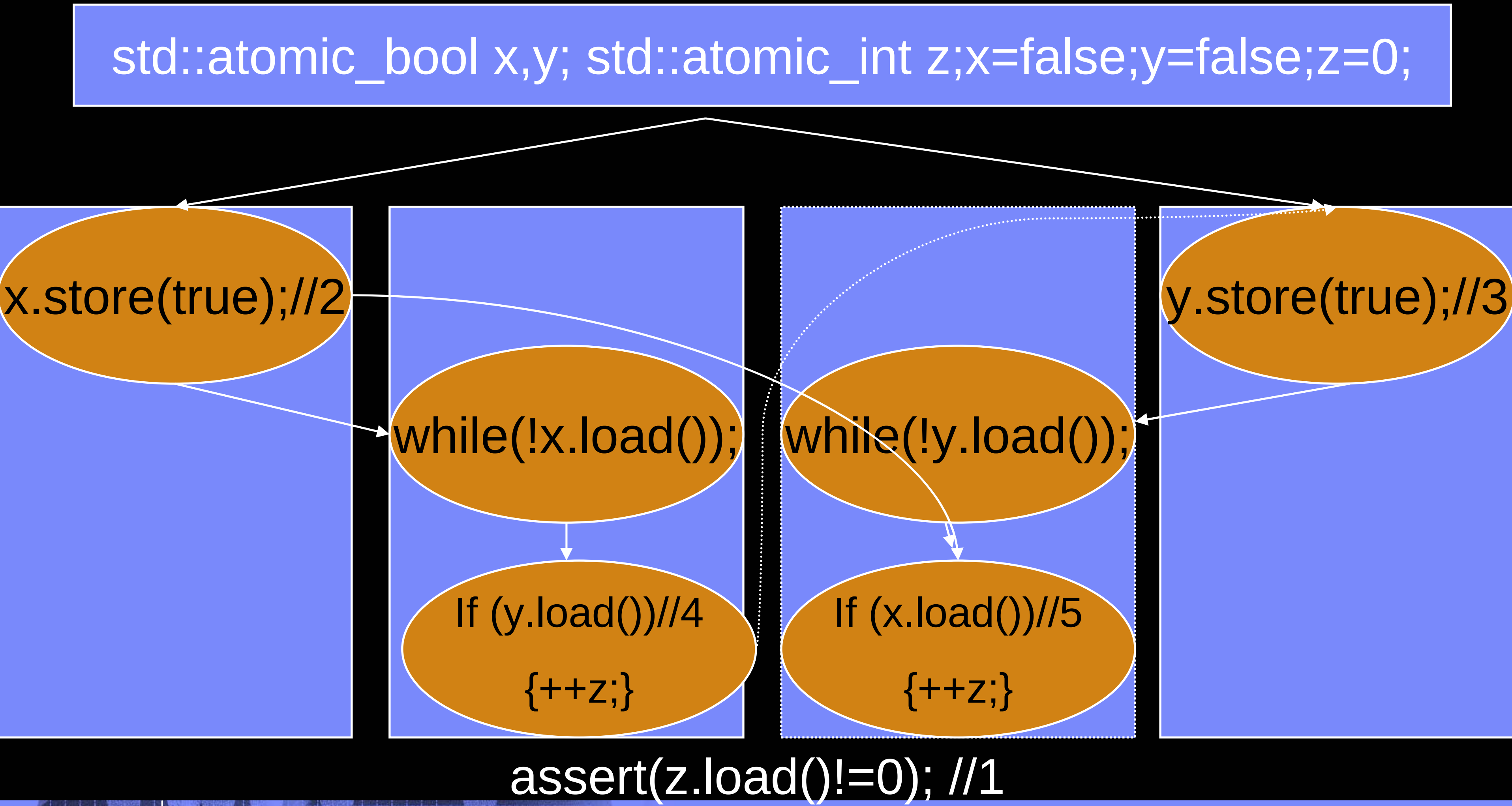
} memory_order;

- **Every atomic operation has a default form, implicitly using `seq_cst`, and a form with an explicit order argument**
- **When specified, argument is expected to be just an enum constant**

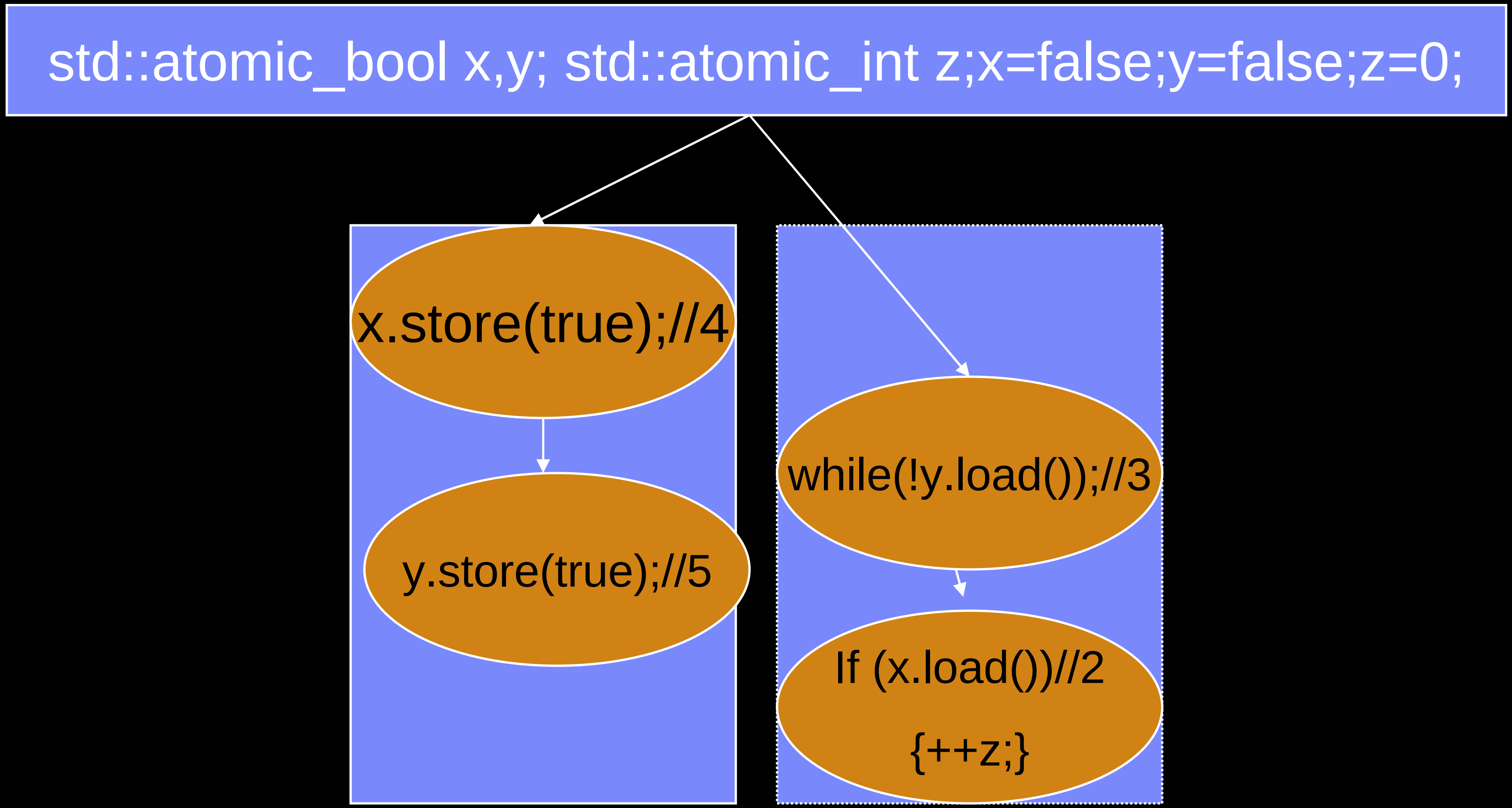
Memory Ordering Constraints

- Sequential Consistency
  - **Single total order for all SC ops on all variables**
  - **default**
- Acquire/Release
  - **Pairwise ordering rather than total order**
  - **Independent Reads of Independent Writes don't require synchronization between CPUs**
- Relaxed Atomics
  - **Read or write data without ordering**
  - **Still obeys happens-before**

SC and happens-before

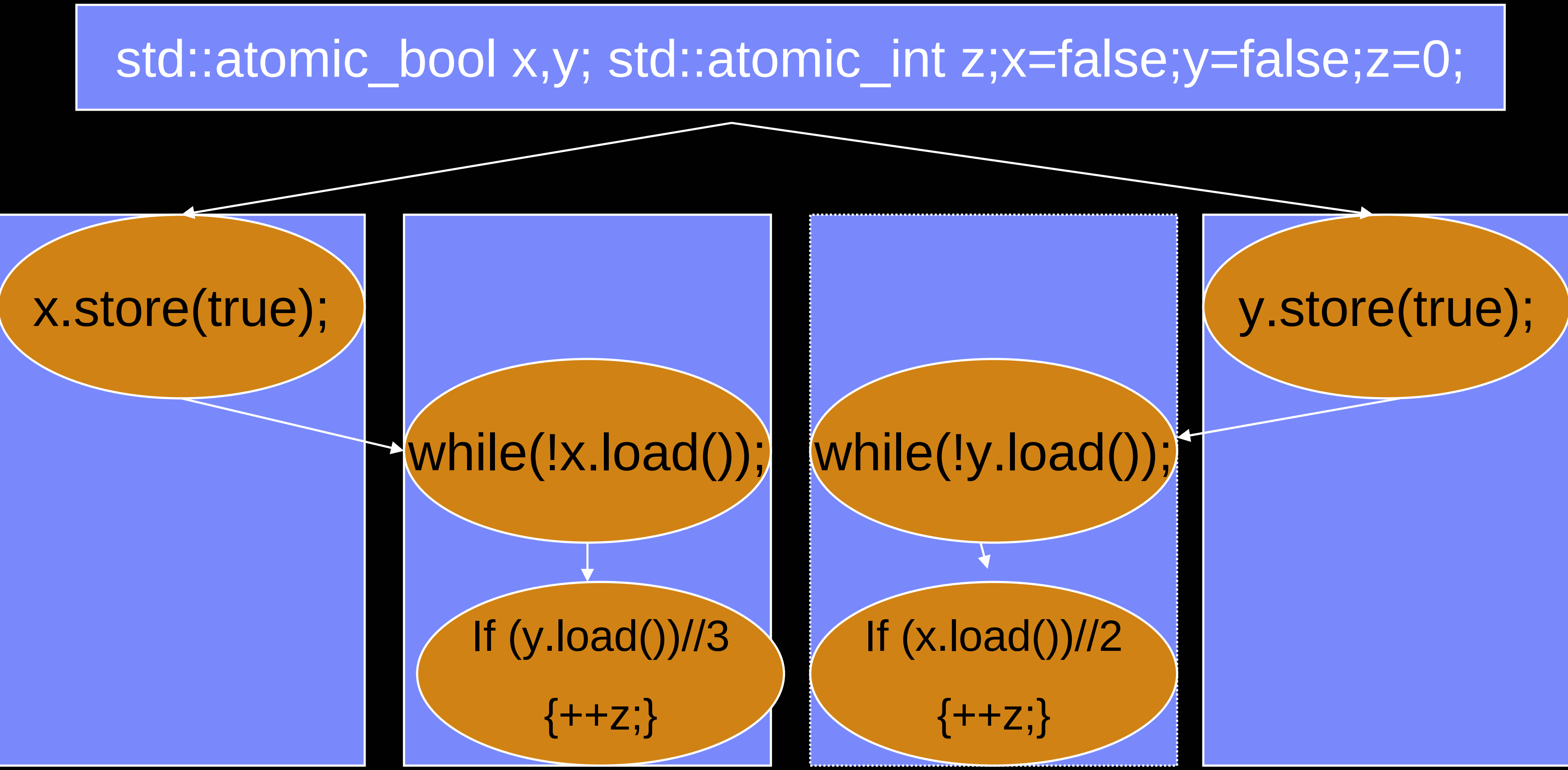


Relaxed and happens-before



assert(z.load()!=0); //1

Acquire-Release with Happens-before



assert(z.load()!=0); //1

Food for thought and Q/A

- **This is the chance to make comments on the C++0x FCD draft through us or the National Body rep:**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3092.pdf
- Memory Model:
  - **http://www.hpl.hp.com/personal/Hans_Boehm/c++mm**
- **Participate and feedback to Compiler**
- **Talk to me at my blog:**
  - http://www.ibm.com/software/rational/cafe/blogs/cpp-standard

My blogs and email address

- michaelw@ca.ibm.com
- Rational C/C++ cafe: http://www.ibm.com/software/rational/cafe/community/ccpp
- My Blogs:
- Parallel & Multi-Core Computing http://www.ibm.com/software/rational/cafe/blogs/ccpp-parallel-multicore
- C++ Language & Standard http://www.ibm.com/software/rational/cafe/blogs/cpp-standard
- Commercial Computing http://www.ibm.com/software/rational/cafe/blogs/ccpp-commercial
- Boost test results http://www.ibm.com/support/docview.wss?rs=2239&context=SSJT9L&uid=swg27006911
- C/C++ Compilers Support Page http://www.ibm.com/software/awdtools/ccompilers/support/
- C/C++ Feature Request Interface http://www.ibm.com/support/docview.wss?uid=swg27005811
- XL Fortran Compiler Support Page http://www.ibm.com/software/awdtools/fortran/xlfortran/support/
- XL Fortran Feature Request Interface http://www.ibm.com/support/docview.wss?uid=swg27005812

Acknowledgement

- **Some slides are borrowed from committee presentations by various committee members, their proposals, and private communication**