Multithreaded C11 & C++11: the Dawn of new Standards (and a preview of upcoming Transactional Memory TS)

Michael Wong
michaelw@ca.ibm.com
IBM Toronto Lab

International Standard Trouble Maker, Chief Cat Herder
IBM and Canadian C++ Standard Committee HoD
OpenMP CEO
Chair of WG21 SG5 Transactional Memory
Director, Vice President of ISOCPP.org

Vice Chair Standards Council of Canada Programming Languages
Acknowledgement and Disclaimer

• Numerous people internal and external, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.

• I even lifted this acknowledgement and disclaimer from some of them.

• But I claim all credit for errors, and stupid mistakes. These are mine, all mine!

• Any opinions expressed in this presentation are my opinions and do not necessarily reflect the opinions of IBM.
IBM Rational Disclaimer

• © Copyright IBM Corporation 2013. All rights reserved. The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. Product release dates and/or capabilities referenced in these materials may change at any time at IBM’s sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way. IBM, the IBM logo, Rational, the Rational logo, Telelogic, the Telelogic logo, and other IBM products and services are trademarks of the International Business Machines Corporation, in the United States, other countries or both. Other company, product, or service names may be trademarks or service marks of others.
C/C++ Café
Boosting Performance, Productivity, and Portability

Welcome to the C/C++ Café

Join
A community of Industry Leaders in C/C++ Technology

Learn
To Take full advantage of the IBM C/C++ compilers

Download
Trials of new technology

Share
Participate in forum discussions. Follow and Respond to Blogs.
Agenda

- C11, C++11, C++14, SG5 TM goals and timelines
- C++ Standard Transactional Memory status
- Multithreading support in C11 and C++11
- The problems of Concurrency before C/C++ 11
- Language support
- Memory Model
- Fragen?
Where were you in 1998?

- Google was incorporated and hires its first employee
- Paypal founded, Amazon buys IMDB
- XML published
- Intel Pentium II: 0.45 GFLOPS
- No SIMD: SSE came in Pentium III
- No GPUs: GPU came out a year later
- The fastest computer was Sandia’s ASCI Red at 1.8 Tflops (9152 cores)
- C++98 became the new JTC1/SC22/WG21 C++ Standard
- A year later, C99 became the WG14 C Standard
C and C++ Standard Progress

All information subject to change without notice

1999 C Std

1998 C++ Std

2001 C TC1

2004 C TC2

2007 C TC3

2005 C++ TR1

2010: FCD published

2011: C11 DIS

2014 C++14 DIS

2011 C++11 FDIS

But only 3 years since C++11

It’s been 11 years since C++98/C99 to C++11/C11!
# ISO/IEC JTC1 ballot stages

<table>
<thead>
<tr>
<th>Stage</th>
<th>IS</th>
<th>TS</th>
<th>Level</th>
<th>Ballot Period</th>
<th>Requirements &amp; Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Proposal</td>
<td>NP</td>
<td>NP</td>
<td>SC22</td>
<td>3 mo</td>
<td>5 NBs agree to actively participate</td>
</tr>
<tr>
<td>30 Committee</td>
<td>CD</td>
<td>PDTS</td>
<td>SC22</td>
<td>2, 3, or 4 mo</td>
<td>Principal comment stage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3-6 mo</td>
<td></td>
</tr>
<tr>
<td>40 Enquiry</td>
<td>DIS</td>
<td>DTS</td>
<td>JTC1 / ITTF</td>
<td>5 mo</td>
<td>2/3 Yes, ≤1/4 No majority</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3-6 mo</td>
<td></td>
</tr>
<tr>
<td>50 Approval</td>
<td>FDIS</td>
<td>—</td>
<td>JTC1 / ITTF</td>
<td>2 mo</td>
<td>No comments 2/3 Yes, ≤1/4 No</td>
</tr>
</tbody>
</table>
C++14 DIS, 8 TS’s under development (by Herb Sutter)
C++14 is approved (photos by Chandler Carruth)
Current Project Details

- **Programming Language C++ IS: Richard Smith.** This is the main C++ Standard project.
- **File System TS: Beman Dawes.** Work based on Boost.Filesystem v3, including file and directory iteration.
- **Library Fundamentals TS: Jeffrey Yasskin.** A set of standard library extensions for vocabulary types like optional<> and other fundamental utilities.
- **Networking TS: Kyle Kloepper.** A small set of network-related libraries including support for network byte order transformation and URIs.
- **Concepts TS: Andrew Sutton.** Extensions for template type checking.
- **Arrays TS: Lawrence Crowl.** Language and library extensions related to arrays, including runtime-sized arrays (aka arrays of runtime bound) and dynarray<>.
- **Parallelism TS: Jared Hoberock.** Initially includes a Parallel STL library with support for parallel algorithms to exploit multiple cores, and vectorizable algorithms to exploit CPU and other vector units.
- **Concurrency TS: Artur Laksberg.** Initially includes library support for executors and non-blocking extensions to std::future. Additionally may include language extensions like await, and additional libraries such as concurrent hash containers and latches.
- **Transactional Memory TS: Michael Wong.** A promising way to deal with mutable shared memory, that is expected to be more usable and scalable than current techniques based on atomics and mutexes.
# Project Timeline

## Current Status: Standard C++ - Mozilla Firefox

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WG21</td>
<td>14082</td>
<td>IS</td>
<td>Programming Language C++</td>
<td>FIS</td>
<td>NP</td>
<td>NP</td>
<td>DTS</td>
<td>NP</td>
<td>DTS</td>
</tr>
<tr>
<td>LEWG</td>
<td>19568</td>
<td>TS</td>
<td>Library Fundamentals</td>
<td>PDTS</td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>EWG/LEWG</td>
<td>19569</td>
<td>TS</td>
<td>Arrays</td>
<td>PDTS</td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG1, Concurrency</td>
<td>19570</td>
<td>TS</td>
<td>Parallelism</td>
<td>PDTS</td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG1, Concurrency</td>
<td>19571</td>
<td>TS</td>
<td>Concurrency</td>
<td>PDTS</td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG2, Modules</td>
<td>18822</td>
<td>TS</td>
<td>File System</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG3, File System</td>
<td>19216</td>
<td>TS</td>
<td>Networking</td>
<td>PDTS</td>
<td>NP</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG4, Networking</td>
<td></td>
<td></td>
<td>Transactional Memory</td>
<td>NP</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG5, Tx Memory</td>
<td>19217</td>
<td>TS</td>
<td>Concepts Lite</td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG6, Numerics</td>
<td></td>
<td></td>
<td>SD.G</td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG7, Reflection</td>
<td></td>
<td></td>
<td>SD.G Feature Testing Recommendations</td>
<td>NP</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG8, Concepts</td>
<td></td>
<td></td>
<td></td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG9, Ranges</td>
<td></td>
<td></td>
<td></td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG10, Feature Test</td>
<td></td>
<td></td>
<td></td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG11, Databases (inactive)</td>
<td></td>
<td></td>
<td></td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG12, U. Behavior</td>
<td></td>
<td></td>
<td></td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
<tr>
<td>SG13, I/O</td>
<td></td>
<td></td>
<td></td>
<td>PDTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
<td>DTS</td>
</tr>
</tbody>
</table>
Agenda

• C11, C++11, C++14, SG5 TM goals and timelines
• C++ Standard Transactional Memory status
• Multithreading support in C11 and C++11
• The problems of Concurrency before C/C++ 11
• Language support
• Memory Model
• Fragen?
Why do we need a TM language?

- TM requires language support
- Hardware here and now
- Multiple projects extend C++ with TM constructs
- Adoption requires common TM language extensions
What is hard about adding TM to C++

• Conflict with C++ 11 memory model and atomics
• Support member initializer syntax
• Support C++ expressions
• Work with legacy code
• Structured block nesting
• Multiple entry and exit from transactions
• Polymorphism
• Exceptions
Project History

• 2008: every other week discussions by Intel, Sun/Oracle, IBM started in July, joined by HP, Redhat, academics
• 2009: Version 1.0 released in August
• 2011: Version 1.1 fixes problems in 1.0, exceptions
• 2012: Brought proposal to C++Std SG1; became SG5, demonstrated use-cases, performance data
• 2013: Presented to Evolution WG as a proposed C++ Technical Specification
• 2014: Approved by Evolution WG; voted in Full group to start New Proposal as a Technical Specification
Active members

- Hans Boehm, hans.boehm@hp.com
- Justin Gottschlich, justin.gottschlich@intel.com
- Victor Luchangco, victor.luchangco@oracle.com
- Jens Maurer, jens.maurer@gmx.net
- Paul McKenney, paulmck@linux.vnet.ibm.com
- Maged Michael, maged.michael@gmail.com
- Mark Moir, mark.moir@oracle.com
- Torvald Riegel, triegel@redhat.com
- Michael Scott, scott@cs.rochester.edu
- Tatiana Shpeisman, tatiana.shpeisman@intel.com
- Michael Spear, spear@cse.lehigh.edu
- Michael Wong, michaelw@ca.ibm.com

Towards a Transaction-safe C++
Standard Library: std::list (N3862)
Commercial Hardware TMs

- Azul Systems’ HTM (phased out?)
- AMD ASF (unknown status)
- Sun’s Rock (cancelled)
- IBM’s Blue Gene/Q (2011)
- Intel’s TSX (code named Haswell) (2012)
- IBM’s zEC12 (2012)
- IBM Power8 (2014)

- HTM will only improve existing STM performance
Commercial/OS Compilers

- Sun Studio (for Rock)
- Intel STM
- IBM AlphaWorks STM (for BG)
- GNU 4.7
- IBM xlc z/OS v1R13 compiler
Design goals

Build on the C++11 specification
  – Follow established patterns and rules
  – “Catch fire” semantics for racy programs

Enable easy adoption
  – Minimize number of new keywords
  – Do not break existing non-transactional code

Have constructs to enable static error detection and runtime selection
  – Ease of debugging is important but so is flexibility

When in doubt, leave choice to the programmer
  – Abort or irrevocable actions?
  – Commit-on-exception or rollback-on-exception?
Locks are Impractical for Generic Programming=
callback

Thread 1:
m1.lock();
m2.lock();
...

Thread 2:
m2.lock();
m1.lock();
...

= deadlock

Easy. Order Locks.
Now let’s get slightly more real:

What about Thread 1 + A thread running f():

What locks does \( x = y \) acquire?
What locks does \( x = y \) acquire?

- Depends on the type \( T \) of \( x \) and \( y \).
  - The author of \( f() \) shouldn’t need to know.
    - That would violate modularity.
  - But lets say it’s `shared_ptr<TT>`.
    - Depends on locks acquired by \( TT \)’s destructor.
    - Which probably depends on its member destructors.
    - Which I definitely shouldn’t need to know.
    - But which might include a `shared_ptr<TTT>`.
      - Which acquires locks depending on \( TTT \)’s destructor.
      - Whose internals I definitely have no business knowing.
      - ...

- And this was for an unrealistically simple \( f() \)!

  - **We have no realistic rules for avoiding deadlock!**
  - In practice: Test & fix?

```
template <class T>
void f(T &x, T y) {
    unique_lock<mutex> _(m2);
    x = y;
}
```
Transactions Naturally Fit Generic Programming Model

- Composable, no ordering constraints

```cpp
f() implementation:
template <class T>
void f(T &x, T y) {
    transaction {
        x = y;
    }
}
```

```cpp
Class implementation:
class ImpT {
    ImpT& operator=(ImpT T& rhs) {
        transaction {
            // handle assignment
        }
    }
};
```

Impossible to deadlock
The Problem

- **Popular belief**: enforced locking ordering can avoid deadlock.

- We show this is essentially impossible with C++ template programming.

- Template programming is pervasive in C++. Thus, template programming needs TM.
Don’t We Know This Already?

• Perhaps, but impact has been widely underestimated.
  – Templates are everywhere in C++.
• Move TM debate away from performance; focus on convincingly correct code.
• Relevant because of C++11 and SG5.
• Generic Programming Needs Transactional Memory by Gottschlich & Boehm, Transact 2013
Conclusion

• Given C++11, generic programming needs TM more than ever.

• To avoid deadlocks in all generic code, even those with irrevocable operations, we need (something like) relaxed transactions.
TM Patterns and Use Cases

• Top four uses cases:
  1. Irregular structures with low conflict frequency
  2. Low conflict structures with high read-sharing and complex operations
  3. Read-mostly structures with frequent read-only operations
  4. Composable modular structures and functions
Current Status of SG5 TM

• EWG Approved to start a NP for a TS 16/6/1/0/0
• LEWG Approved 8/3/2/0/0
• Formal motion for New Proposal TS approved INCITS: 25/0/0 and ISO: 7/0/0
  – Based on N3919 as indicated content
  – 3 sets of Balloting for 12 months to become and official TS
• Continue telecon every other week to create a first TS Working Draft for Rapperswil, Switzerland
• Aimed for Final DTS for 2015
• Continue working on enhancements for further TS
Support for TM in C++ std library

• enable users to use transactional constructs in the first TS delivery of SG5
• Started with std::list
• Make it transaction-safe
  – Enables use with atomic blocks
• Open source collaboration welcome on github
  – https://github.com/mfs409/tm_stl
1 construct for transactions
1. Compound Statements
2 Keywords for different types of TX
atomic_noexcept | atomic_commit | atomic_cancel
    {<compound-statement> } 

synchronized {<compound-statement> }

1 Function/function pointer keyword
transaction_safe
- must be a keyword because it conveys necessary semantics on type
1 Function/function pointer attribute
[[transaction_unsafe]]
- provides static checking and performance hints, so it can be an attribute
What is transaction-safe?

• From N3919, what is a transaction-safe operation?
  – Operations in which system can guarantee atomicity

• excluding:
  – Access to volatile data
  – Assembly instructions
  – Calls to functions that violate atomicity

• Examples of functions that violate atomicity
  – Synchronization: operations on locks/ mutexes and C++11 atomics
  – Certain I/O functions

—More info in:
atomic Examples

class Account {
  int bal;
public:
  Account(int initbal) { bal = initbal; }
  void deposit(double x) {
    atomic_noexcept {
      this.bal += x;
    }
  }
  double balance() { return bal; }
};

void transfer(Account a1, a2; int x;) {
  atomic_noexcept {
    a1.deposit(-x);
    a2.deposit(x);
  }
};

Towards a Transaction-safe C++
Standard Library: std::list (N3862)
int i = 0;
void f() {
  transaction_relaxed {
    if (unlikely_condition)
      std::cerr << "oops" << std::endl;
    ++i;
  }
}
Agenda

• C11, C++11, C++14, SG5 TM goals and timelines
• C++ Standard Transactional Memory status
• Multithreading support in C11 and C++11
• The problems of Concurrency before C/C++ 11
• Language support
• Memory Model
• Fragen?
Is this legal C++03 syntax?

```cpp
template<class T> using Vec = vector<T,My_alloc<T>>;
Vec<double> v = { 2.3, 1.2, 6.7, 4.5 };
//sort(v);
for(auto p = v.begin(); p!=v.end(); ++p)
    cout << *p << endl;
```
```cpp
#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
}
```
Can you do this with TLS before C11/C++11?

```c++
extern std::string f(); // returns "Hello"
from another TU
std::string foo(std::string const& s2) {
  __thread std::string s=f();
  s+=s2;
  return s;
}
```
Is this valid C++ today? Are these equivalent?

```cpp
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_acquire) != 1)
  // or: while (y.load() != 1)
    continue;
  assert(x == 17);
```

```cpp
int x = 0;
atomic<int> y = 0;

Thread 1:
  x = 17;
  y = 1;

Thread 2:
  while (y != 1)
    continue;
  assert(x == 17);
```
C++11, C11

• C++0x: Codename for the planned new standard for the C++ programming language
  – Will replace existing ISO/IEC 14882 standard published in 1998 (C++98) and updated in 2003 (C++03)
  – Many new features to core language
  – Many library features: most of C++ Technical Report 1 (TR1)
  – FDIS in March 2011
  – X=A,B,C,D,E,F?
  – C++11 ratified Dec 2012

• C1x: Codename for the planned new standard for the C programming language
  – Will replace existing ISO/IEC 9899 standard published in 1999
  – DIS in March 2011
  – C11 ratified Dec 2012
Status of Language Standards as of 1H 2013

• C11 ratified Dec 2011,
  – starting work on next C

• C++11 ratified Dec 2011,
  – starting work on C++2014/2017/2022
  • Much more advanced concurrency

• Creation of isocpp.org Foundation to promote Std C++ and centralize latest knowledge
  • Michael Wong is Founding Director & VP
  • IBM is Founding Member
Organization & Sponsors

• Three goals:
  – Handle money: Pay for site development and maintenance by accept funds from sponsors (but not general public).
  – Hold copyright/license: We need Pearson to be able to publish an edited version of the wiki FAQ as a \textit{C++ FAQs 3e} book/e-book.
  – Have a clear board: We need to make it clear “who this is,” who is involved and directing the effort.

• “Standard C++ Foundation” 501c(6)
  – Founding sponsors: Some subset of \{ AAPL, Boost/BoostPro, Bloomberg, C&B, FB, GOOG, IBM, INTL, MSFT, TAMU, \ldots \}
    • Gold: $10K/year
    • Silver:$5K/year
    • Bronze:$1K?year
  – Board of directors:
    • Chandler Carruth (Google) [Treasurer]
    • Beman Dawes (Boost)
    • Stefanus Du Toit (Intel) [Secretary]
    • Bjarne Stroustrup (Texas A&M University)
    • Herb Sutter (Microsoft) [Chairman, President]
    • Michael Wong (IBM) [Vice President]
C++11 Library

• Start with original C++98 library
  – Improved performance with rvalue reference
  – Used variadic templates to improve compile time
  – Potential binary incompatibility with C++98 library strings
  – Reference counting not allowed

• Added 13/14 TR1 libraries
  – Reference wrapper, smart ptrs, return type determination, enhanced
    member pointer adapter, enhanced binder, generalized functors, type
    traits, random numbers, tuples, fixed size array, hash tables, regular
    expressions, C99 cmpat

• Added threading, unique_ptr, forward_list, many
  new algorithms
Sum of all things C11

- Alignment specification
  - __Alignas specifier
  - Alignof operator
  - Aligned_alloc function
  - <stdalign.h> header
- __Noreturn function specifier
- Type-generic expressions
  - __Generic keyword
- Multithreading support
  - __Thread_local storage class specifier
- Improved unicode (UTF16/32/8)
- Remove gets
- Macros for querying subnormal floating point number and number of digits the type is able to store
- Anonymous structs/unions
- Static assertions
- Exclusive create-and-open mode for fopen
- Quick_exit
- Macros for construction of complex values

- Optional features new from C99
  - Analyzability (Annex L)
  - Bounds Checking (Annex K)
  - Multithreading <threads.h>
  - Atomic primitives and types
    Mstdatomic.h> and _Atomic type qualifier

- Optional features optional in C99
  - IEC 60559 floating point arithmetic (Annex F)
  - IEC 60559 compatible complex arithmetic (Annex G)

- Optional features mandatory in C99
  - Complex types (mandatory for hosted)
  - Variable length arrays
Removed or Deprecated features

C++
• Auto as a storage class
• Export semantics
• Register semantics
• Exception specification
• Auto_ptr
• Bind1st/bind2nd
• bool++
• See Clause D

C
• Gets
C++11 land:
http://fearlesscoder.blogspot.ca/2012/01/c11-lands.html
C++11

Variables
- Static
- Dynamic
- Declaration
- Definition
- Initialization

Constants
- Lambda expressions
- Closures
- Default capture mode
- Lambda expressions
- Mutables

Functors
- Functions
- Operators

Classes
- Constructors
- Destructors
- Data members
- Bit-fields
- Virtual
- Final
- Override

Methods
- Pure virtual
- Defaulted
- Deleted
- Const

Static members
- Public
- Protected
- Private

Instantiation
- Friend
- Friendship
- Forward declaration
- Struct

Types
- Arrays
- Unions
- References
- Pointers
- Const
- Volatile
- Complete
- Alignment
- Conversions
- Typedef
- Auto
- Templates
- Arguments
- Template template arguments
- Specialization
- Partial specialization
- Variadic
- Sizeof
- Function templates
- Argument deduction
- Aliases
- Name resolution
- Instantiation
- Extern
What changed in C++11

### C++11 STL additions

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>all_of(iterator first, iterator last, predicate)</code></td>
<td>unordered_set&lt;T&gt; contains at most one of each value and provides fast retrieval of values; supports forward iterators</td>
</tr>
<tr>
<td><code>any_of(iterator first, iterator last, predicate)</code></td>
<td>unordered_multiset&lt;T&gt; supports equivalent values (possibly with multiple copies of the same value) and provides fast retrieval of the values; supports forward iterators</td>
</tr>
<tr>
<td><code>none_of(iterator first, iterator last, predicate)</code></td>
<td>unordered_map&lt;Key, T&gt; hash table; contains at most one of each key value, supports forward iterators</td>
</tr>
<tr>
<td><code>find_if_not(iterator first, iterator last, predicate)</code></td>
<td>unordered_multimap&lt;Key, T&gt; hash table; supports equivalent key (can contain multiple copies of the same key value); supports forward iterators</td>
</tr>
<tr>
<td><code>find_if(iterator first, iterator last, predicate)</code></td>
<td>ordered set&lt;T&gt; contains at most one of each value and provides fast retrieval of values; supports forward iterators</td>
</tr>
<tr>
<td><code>find(iterator first, iterator last, value)</code></td>
<td>ordered_multiset&lt;T&gt; supports equivalent values (possibly with multiple copies of the same value) and provides fast retrieval of the values; supports forward iterators</td>
</tr>
<tr>
<td><code>remove</code></td>
<td>ordered_map&lt;Key, T&gt; hash table; contains at most one of each key value, supports forward iterators</td>
</tr>
<tr>
<td><code>remove_copy</code></td>
<td>ordered_multimap&lt;Key, T&gt; hash table; supports equivalent key (can contain multiple copies of the same key value); supports forward iterators</td>
</tr>
<tr>
<td><code>remove_copy_if</code></td>
<td>pair&lt;iterator, iterator&gt; partition_copy(iterator first, iterator last, iterator before=True, iterator after=True)</td>
</tr>
<tr>
<td><code>erase</code></td>
<td>ordered set&lt;T&gt; contains at most one of each value and provides fast retrieval of values; supports forward iterators</td>
</tr>
<tr>
<td><code>erase_copy</code></td>
<td>ordered_map&lt;Key, T&gt; hash table; contains at most one of each key value, supports forward iterators</td>
</tr>
<tr>
<td><code>erase_copy_if</code></td>
<td>equivalent set&lt;T&gt; supports equivalent values (possibly with multiple copies of the same value) and provides fast retrieval of the values; supports forward iterators</td>
</tr>
<tr>
<td><code>insert</code></td>
<td>equivalent multiset&lt;T&gt; supports equivalent values (possibly with multiple copies of the same value) and provides fast retrieval of the values; supports forward iterators</td>
</tr>
<tr>
<td><code>insert_copy</code></td>
<td><code>insert_copy_if</code></td>
</tr>
<tr>
<td><code>insert_if</code></td>
<td><code>before_begin</code>, <code>begin</code>, <code>end</code></td>
</tr>
<tr>
<td><code>erase</code></td>
<td><code>erase_copy</code>, <code>erase_copy_if</code></td>
</tr>
<tr>
<td><code>erase_copy_if</code></td>
<td><code>erase_if</code></td>
</tr>
<tr>
<td><code>erase_if</code></td>
<td><code>erase_if</code></td>
</tr>
</tbody>
</table>

---

*Want to be a C++11 expert? Check out cpprocks.com*
Concurrency in C11/C++11

• C99/C++98/03: does not have concurrency
• C++11 is in Final Draft International Standard on 2011
• C11 is in Draft International Standard in 2011
• C++11 have multithreading support
  – Memory model, atomics API
  – Language support: TLS, static init, termination, lambda function
  – Library support: thread start, join, terminate, mutex, condition variable
  – Advanced abstractions: basic futures, thread pools
• C11 will have similar memory model, atomics API. TLS, static init/termination
  – Some minor differences like __Atomic qualifier
Introduction to concurrency

• Why do we need to standardize concurrency
  – Multi-core processors
  – Solutions for very large problems
  – Internet programming

• Standardize existing practice
  – C++ threads=OS threads
  – shared memory
  – Loosely based on POSIX, Boost thread
  – Does not replace other specifications
    • MPI, OpenMP, UPC, autoparallelization
  – Can help existing advanced abstractions
    • TBB, PPL, Cilk,
Concurrenty core/library

• Core: what does it mean to share memory and how it affects variables
  – TLS
  – Static duration variable initialization/destruction
  – Memory model
  – Atomic types and operations
    • Lock-free programming
  – Fences
  – Dependence based Ordering

• Library
  – How to create/synchronize/terminate threads,
  – Thread, mutex, locks
    • RAII for locking, type safe
  – propagate exceptions
  – A few advanced abstraction
    • Async(), promises and futures
What we got

• Low level support to enable higher abstractions
  – Thread pools, TM
• Ease of programming
  – Writing correct concurrent code is hard
  – Lots of concurrency in modern HW, more than you imagine
• Portability with the same natural syntax
  – Not achievable before
• Uncompromising Performance
• Stable memory model
• System level interoperability
  • C++ shares threads with other languages
What we did not get

• All the nifty, higher parallel abstractions
  – TM, thread pools, futures, parallel STL
• Complete Compatibility between C and C++
• Total insolation from programmer mistakes
The grand scheme of Concurrency

<table>
<thead>
<tr>
<th>Asynchronous Agents</th>
<th>Concurrent collections</th>
<th>Mutable shared state</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>summary</strong></td>
<td>tasks that run independently and communicate via messages</td>
<td>operations on groups of things, exploit parallelism in data and algorithm structures</td>
</tr>
<tr>
<td><strong>examples</strong></td>
<td>GUI, background printing, disk/net access</td>
<td>trees, quicksorts, compilation</td>
</tr>
<tr>
<td><strong>key metrics</strong></td>
<td>responsiveness</td>
<td>throughput, many core scalability</td>
</tr>
<tr>
<td><strong>requirement</strong></td>
<td>isolation, messages</td>
<td>low overhead</td>
</tr>
<tr>
<td><strong>today’s abstractions</strong></td>
<td>thread, messages</td>
<td>thread pools, openmp</td>
</tr>
<tr>
<td><strong>future abstractions</strong></td>
<td>futures, active objects</td>
<td>chores, parallel STL, PLINQ</td>
</tr>
</tbody>
</table>
Memory Model and Consistency model, a quick tutorial

• Sequential Consistency (SC)

**Sequential consistency was originally defined in 1979 by Leslie Lamport as follows:**

• “... the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program”

• But chip/compiler designers can be annoyingly helpful:
  • It can be more expensive to do exactly what you wrote.
  • Often they’d rather do something else, that could run faster.
Sequential Consistency: a tutorial

• The semantics of the single threaded program is defined by the program order of the statements. This is the strict sequential order. For example:

```
x = 1;

r1 = z;

y = 1;

r2 = w;
```
Sequential Consistency for program understanding

• Suppose we have two threads. Thread 1 is the sequence of statement above. Thread 2 is:

Thread 1:
\[
\begin{align*}
x &= 1; \\
r1 &= z; \\
y &= 1; \\
r2 &= w;
\end{align*}
\]

Thread 2:
\[
\begin{align*}
w &= 1; \\
r3 &= y; \\
y &= 1; \\
r4 &= x;
\end{align*}
\]

(All variables are initialized to zero.)

• 2 of 4! Possible interleavings:

\[
\begin{align*}
x &= 1; &\quad x &= 1; \\
w &= 1; &\quad w &= 1; \\
r1 &= z; &\quad r3 &= y; \\
r3 &= y; &\quad z &= 1; \\
y &= 1; &\quad r4 &= x; \\
z &= 1; &\quad r1 &= z; \\
r2 &= w; &\quad y &= 1; \\
r4 &= x; &\quad r2 &= w;
\end{align*}
\]
Now add fences to control reordering

Thread 1:
\[
\begin{align*}
x &= 1; \\
r1 &= z; \\
fence(); \\
y &= 1; \\
r2 &= w; \\
\end{align*}
\]
Is \(r3 = 1\) and \(r4 = 1\) possible?
Is \(r1 = 1\) and \(r2 = 1\) possible?

Thread 2:
\[
\begin{align*}
w &= 1; \\
r3 &= y; \\
fence(); \\
z &= 1; \\
r4 &= x; \\
\end{align*}
\]
Agenda

- C11, C++11, C++14, SG5 TM goals and timelines
- C++ Standard Transactional Memory status
- Multithreading support in C11 and C++11
- The problems of Concurrency before C/C++ 11
- Language support
- Memory Model
- Fragen?
Memory Model and instruction reordering

- Definitions:
- Instruction reordering: When a program executes instructions, especially memory reads and writes, in an order that is different than the order specified in the program’s source code.
- Memory model: Describes how memory reads and writes may appear to be executed relative to their program order.
- Affects the valid optimizations that can be performed by compilers, physical processors, and caches.
Memory Model and Consistency model

- Sequential Consistency (SC)

**Sequential consistency was originally defined in 1979 by Leslie Lamport as follows:**

- “... the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program”
- But chip/compiler designers can be annoyingly helpful:
  - It can be more expensive to do exactly what you wrote.
  - Often they’d rather do something else, that could run faster.
Problems for concurrency in 2005

• How much and what kind of reordering is allowed?
  – For programmer understanding?
  – For better performance

• What motivates each paragraph of clause 1.10, and Chap 29 of the C++0x Standard
  1. The entire software/hardware stack
  2. Languages and compilers
  3. Volatile
  4. Compiler-generated data races
  5. C++ destructors
  6. Thread libraries
  7. C++ Standard Library
  8. Thread-safe C libraries
  9. Broken C++ idioms
Problem 1: Hardware at the bottom of the stack

• Everything from threads implementations to user code depends on memory consistency/ordering:
• Canonical Example (assume all init with 0, all shared variables):
  Thread 1
  x=1;
r1=y;  //reads 0
  Thread 2
  y=1;
r2=x;  //reads 0
Can both r1 and r2 be 0?
• Intuitively (or under sequential consistency) no; some thread executes first.
• In practice, yes; compilers, thread library and hardware can reorder.
• Most hardware will allow this outcome because they have write buffers!
Hardware at the bottom of the stack

- The hardware doesn’t implement what we tell it is fundamentally a problem
  - If we understand HW rules and can use them to implement a usable programming model
- Widely held belief
  - Weaker memory models (e.g. allowing this) is fine, since
    - We only pay for ordering (special fence instructions) when needed
      - Should be cheaper
    - Fence instructions get us sequential consistency exactly when we need it
Transformations: in the name of speed

• Reordering, invention, removal

• Entire stack:
  – Source code
  – Compiler
  – Hardware
  – Cache
  – Execution
Dekker’s and Peterson’s Algorithms

- Consider (flags are shared and atomic, initially zero):
  
  **Thread 1:**
  
  ```c
  flag1 = 1; // a: declare intent to enter
  if( flag2 != 0 ) { ... } // b: detect and resolve contention
  // enter critical section
  ```

  **Thread 2:**
  
  ```c
  flag2 = 1; // c: declare intent to enter
  if( flag1 != 0 ) { ... } // d: detect and resolve contention
  // enter critical section
  ```

- Could both threads enter the critical region?
- Maybe: If a can pass b, and c can pass d, we could get b->d->a->c.
- Solution 1 (good): Use a suitable atomic type (e.g., Java/.NET “volatile”, C++0x std::atomic<>) for the flag variables.
- Solution 2 (good?): Use system locks instead of rolling your own.
- Solution 3 (harder but fast): Write a memory barrier after a and c.
What really happens when you are not looking

Processor 1

flag1 = 1; (1)
if( flag2 != 0 ) {...}; (3)

Write 1 to flag1 (sent to store buffer)

Read 0 from flag2 (read allowed to pass buffered store to different location)

Flush buffered store to flag1

Store Buffer

Processor 2

flag2 = 1; (2)
if( flag1 != 0 ) {...} (4)

Write 1 to flag2 (sent to store buffer)

Read 0 from flag1 (read allowed to pass buffered store to different location)

Flush buffered store to flag2

Store Buffer

Global Memory
Memory Ordering and fences in 2006

• Some architectures have underspecified memory ordering
  – Confusion
  – Interesting consequences, X86
    • Gcc __sync_synchronize() full memory barrier erroneously generates no-op
    • P4 lfence, sfence instructions appeared to be no-ops in most user code, but the loads and stores are already ordered
Performance

• Performance of fences and syncs was often neglected
  – More than 100 cycle, best case on P4
  – Encourages
    • Clever sync avoidance techniques = bugs
  – Can easily be much more expensive than sequential consistency everywhere (PA-RISC)
• Some Memory models in which it appeared that fences could not enforce sequential consistency
  – This means Java memory model is not really implementable
Independent Reads with Independent Writes

• x, y init to 0, add fences between every instruction

Thread 1      Thread 2      Thread 3      Thread 4
x=1;      y=1;    r1=x;(1)    r3=y;(1)
fence;    fence;    fence;
r2=y;(0)  r4=x;(0)

x set first!  y set first!

Can this be both true?
Architecture in late 2007

- Intel and AMD published Memory models
- IBM has published PowerPC Atomic operations
- Can get sequential consistency examples like the preceding one on X86
  - But stores to x and y in T1 and T2 need to be implemented with xchg
    - Many JVMs will need to be fixed because they didn’t know this rule when they were written
- Most other vendors are paying attention
Problem 2: Languages and Compilers

• Programming rules are unclear, some languages more then others
  – Java is in best shape since fix in 2005
  – .Net and OpenMP need to be clearer
  – C99 and C++98 has no rules! Other then volatile, sequence points
  – C++11 will hopefully be the best
C++11 programming rules:

- No simultaneous access from two threads to ordinary shared variables if one access is a write, i.e. data races are outlawed
  - Posix C has this rule too, explored in Sarita Adve’s Ph.D thesis
  - This rule dates back to at least Ada83
- No Data races solves many problems:
  - Can’t tell whether compiler reorders ordinary memory operations
    - If you could tell, observing thread would race with updating thread
    - Can’t tell whether hardware reorders memory operations (as long as locks are handled)
  - C/C++ compilers may rely on the absence of asynchronous changes. This may have weird side effects
Consequence of “no async changes” compiler assumption:

```c
unsigned x;
If (x<3) {
    // async x change
    switch (x) {
        case 0: ...
        case 1: ...
        case 2: ...
    }
}
```

- Assume switch statement compiled as branch table
- May assume x is in range
- Async change to x causes wild branch
  - Not just wrong value
C and C++ thread realities before standardization

• Common attitude that data races aren’t so bad
  – Frequently used idioms rely on benign data races:
    • Approximate counters sometimes without locking an update, and read asynchronously
    • Double-checked locking: lazy init that reads flag outside of critical section
  – Or nonportable atomic (interlocked, __sync) operations
    • Eg. Reference counting
  – Not well-defined, and read accesses generally appear as data races to compiler
    • This can result in crashed, reads of half-updated values, uninitialized data
    • But locks are expensive enough that this is often impractical to avoid
Problem 3: A Volatile Market

• 2005 java:
  
  volatile x_init:
  x=1;
  //possible fence here
  x_init=true;
  This guarantees that x becomes visible to other threads before x_init.
Volatile

- OpenMP 2.5 and 3.0 Revision 11 Clause 1.4, Pg 15 Line 26-31
  The volatile keyword in the C and C++ languages specifies a consistency mechanism that is related to the OpenMP memory consistency mechanism in the following way: a reference that reads the value of an object with a volatile-qualified type behaves as if there were a flush operation on that object at the previous sequence point, while a reference that modifies the value of an object with a volatile-qualified type behaves as if there were a flush operation on that object at the next sequence point.
POSIX Pthread C binding

• David Butenof:
  “volatile ... provide[s] no help whatsoever in making code ‘thread safe’”
C++ 11 WP solution

- Concurrent access to special atomic objects is allowed
  - atomic <int>
- Really just a communication issue
- Eliminates all benign data races in C++
- C++ volatile continues to have nothing to do with threads
- Java volatile = C++ atomic
- C++ volatile != Java volatile
Problem 4: Compiler generated data races

• Compiler may generate code that adds data races!
  – When small struct fields are updated
  – Optimizations leading to spurious writes of old values
  – C++03 allows this!
Adjacent bitfield memory

- Given a global s of type struct { int a:9; int b:7; }:
  
  **Thread 1:**
  {lock<mutex> hold( aMutex );s.a = 1;}
  Thread 2:
  {lock<mutex> hold( bMutex );s.b = 1;}

- Is there a race? Yes in C++0x, in pthreads
  - It may be impossible to generate code that will update the bits of a without updating the bits of b, and vice versa.
  - C++0x will say that this is a race. Adjacent bitfields are one “object.”
Adjacent scalar types

- What about two global variables char c; and char d;?
  Thread 1:
  {lock<mutex> hold( cMutex );c = 1;}
  Thread 2:
  {lock<mutex> hold( dMutex );d = 1;}
- Is there a race? No ideally and in C++0x, but maybe today in pthreads
- Say the system lays out c then d contiguously, and transforms “d = 1” to:
  char tmp[4]; // 32-bit scratchpad
  memcpy( &tmp[0], &c, 4 ); // read 32 bits starting at c
  tmp[2] = 1; // set only the bits of d
  memcpy( &c, &tmp[0], 4 ); // write 32 bits back
- Oops: Thread 2 now silently also writes to c without holding cMutex.
Other things that go Bump in the night

• There are many transformations. Here are two common ones.
• Speculation:
  – Say the system (compiler, CPU, cache, ...) speculates that a condition may be true (e.g., branch prediction), or has reason to believe that a condition is often true (e.g., it was true the last 100 times we executed this code).
  – To save time, we can optimistically start further execution based on that guess. If it’s right, we saved time. If it’s wrong, we have to undo any speculative work.
• Register allocation:
  – Say the program updates a variable \( x \) in a tight loop. To save time: Load \( x \) into a register, update the register, and then write the final value to \( x \).
• Key issue: The system must not invent a write to a variable that wouldn’t be written to (in an SC execution).
• If the programmer can’t see all the variables that get written to, they can’t possibly know what locks to take.
Eliding locks

- Consider (where x is a shared variable, cond does not change):
  
  ```
  if( cond )lock x
  ...
  if( cond )use x
  ...
  if( cond )unlock x
  ```

- Q: Is this pattern safe?
- A: In theory, yes. In reality, maybe not...
Write speculation

• Consider (where x is a shared variable):
  if( cond ) x = 42;

• Say the system (compiler, CPU, cache, ...) speculates (predicts, guesses, measures) that cond (may be, will be, often is) true. Can this be transformed to:
  r1 = x; // read what’s there
  x = 42; // perform an optimistic write
  if( !cond) // check if we guessed wrong
      x = r1; // oops: back-out write is not SC

• In theory, No... but on some implementations, Maybe.
  – Same key issue: Inventing a write to a location that would never be written to in an SC execution.
  – If this happens, it can break patterns that conditionally take a lock.
Lessons learned

• All bets are off in a race:
  – Prefer to use locks to avoid races and nearly all memory model weirdness, despite the flaws of locks. (In the future: TM?)
  – Avoid lock-free code. It’s for wizards only, even using SC atomics.
  – Avoid fences even more. They’re even harder, even full fences.

• Conditional locks:
  – Problem: Your code conditionally takes a lock, but your system changes a conditional write to be unconditional.
  – Option 1: In code like we’ve seen, replace one function having a doOptionalWorkflag with two functions (possibly overloaded):
    • One function always takes the lock and does the x-related work.
    • One function never takes the lock or touches x.
  – Option 2: Pessimistically take a lock for any variables you mention anywhere in a region of code.
  – Even if updates are conditional, and by SC reasoning you could believe you won’t reach that code on some paths and so won’t need the lock.
C++11 WP solution:

- Subject to “no data races” rule:
  - Each update affects a “memory location”
    - Scalar value, or contiguous sequence of bitfields
  - Define exactly which assignments can be “seen” by each reference to a memory location
  - For ordinary (non-atomic) references, there must be exactly one, for atomics there can be several
  - A reference to x.d after completion of both threads must see a value of 1
  - The preceding implementation of bit-field assignments is incorrect
  - Assignments s.a and s.b bitfields may still interfere
Non-terminating loops

• Some kinds of code hoisting are problematic.
• Stores may not be advanced across potentially nonterminating loops.
• Example:
  for (T*p = q; p != 0; p = p -> next) ++count;
  x = 42;
• Uncommon? But analysis is commonly wrong.
Subtle implication of the C++11 rule

- No speculative writes
  ```c
  int count; // global, may be shared between threads
  for (p=q; p!=0; p=p->next)
    if (p->data>0) ++ count;
  ```
- Cannot transform into
  ```c
  int count; // global, may be shared
  reg=count;
  for (p=q; p!=0; p=p->next)
    if (p->data>0) ++ reg;
  count=reg; // may spuriously assign to count
  ```
Consequence

• Outlaws some useful optimizations,
• Gives programmer a simple and consistent story
• Prevents really mysterious compiler-introduced program bugs
• Outlawed optimizations can often be replaced by others
Problem 5: C++ destructors

• We just needed assignments to break things

- Even standard library is unsafe to use after `exit()`
- except that threads may return after `main()` calls `exit()`
C++11 WP solution

- Only partial solution:
  1. Shutdown all threads before process exit
     - Hard if they are waiting for I/O
  2. OR execute only special cleanups before exit (but not destructors) A way to kill a process without making libraries unusable, i.e. without running static destructors.
- The former is hard:
  - Need to shut down threads blocked on I/O
  - Tried and failed to get something that played with Posix cancellation.
- Quick_exit() gives us the latter.
  - Something between exit() and _exit()
Problem 6: Thread libraries

• Have to limit reordering of memory operations with respect to synchronization operations:
  lock();
  tmp=x;
  x=tmp+1;
  unlock();

• Done in 2 ways
  – Compiler treats synchronization functions as opaque
    • As though they might change x
  – Synchronization routines contain expensive fence
    tmp=x;
    lock();
    x=tmp+1;
    unlock();
What reordering should we allow?

- Reordering of memory operations with respect to critical sections:

<table>
<thead>
<tr>
<th>Expected(&amp;java)</th>
<th>Naive pthreads</th>
<th>Optimized pthreads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unlock()</td>
<td>Unlock()</td>
<td>Unlock()</td>
</tr>
<tr>
<td>Lock()</td>
<td>Lock()</td>
<td>Lock()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trylock, why pthread doesn’t allow expected reordering

• Some awful code would break (Don’t write code like this!):

Thread 1                              Thread 2
X=42;                                 while (trylock() == SUCCESS)
lock();                                          unlock();
assert (x==42);

• Reordering thread 1 statements is wrong
  – Can’t move into a lock

• Only recognized recently
Some open source pthread lock implementations 2006

<table>
<thead>
<tr>
<th>PPC mutex, spin</th>
<th>Itanium x86 mutex</th>
<th>Itanium x86 spin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock()</td>
<td>Lock()</td>
<td>Lock()</td>
</tr>
<tr>
<td>Unlock()</td>
<td>Unlock()</td>
<td>Unlock()</td>
</tr>
</tbody>
</table>

**FreeBSD spin**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock()</td>
</tr>
<tr>
<td>Unlock()</td>
</tr>
</tbody>
</table>
C++11 WP solution:

• Movement into critical section is allowed in both directions
• Trylock() may fail even if lock is available
  – Problematic examples are now clearly incorrect, achieved our goal
  – But library vendors are informally discouraged from taking advantage of this
    • Worsens performance
• Allows the standard to guarantee that memory operation reordering is invisible for data-race-free programs
  – That don’t use some low level library facilities
  – Such data-race-free programs behave as though thread steps are simply interleaved
Problem 7: C++ Standard Libraries

• General Wisdom about locks in container libraries
  – Lock in the client
  – Only client knows about sharing the right granularity
  – Unexpected library-based locking causes:
    • Performance problems and deadlocks

• But
  – Original Java collections like vectors are synchronized
    • Including individual element access
  – As is Posix putc
    • But not C++ STL containers
C++11 solution

• Follow de facto STL convention
  – Containers do not visibly acquire locks by default
  – Containers behave like scalars:
    • Two operations on a container conflict if one of them logically updates the container.
    • Allocation doesn't count as update.
    • User-invisible updates require internal locking.
    • Other locking is the clients responsibility.
  – This seems to be the de facto standard.
    • except for I/O?
  – Basic_string and reference counting?
    • ABI change?
Problem 8: thread –safe C library

- Four levels of MT safe attributes for library interfaces.
- 1) Unsafe
  - Contains global and static data that are not protected. User should make sure only one thread at time to execute the call.

<table>
<thead>
<tr>
<th>Unsafe Function</th>
<th>Reentrant counterpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctime</td>
<td>ctime_r</td>
</tr>
<tr>
<td>localtime</td>
<td>localtime_r</td>
</tr>
<tr>
<td>asctime</td>
<td>asctime_r</td>
</tr>
<tr>
<td>gmtime</td>
<td>gmtime_r</td>
</tr>
<tr>
<td>ctermid</td>
<td>ctermid_r</td>
</tr>
<tr>
<td>getlogin</td>
<td>getlogin_r</td>
</tr>
<tr>
<td>rand</td>
<td>rand_r</td>
</tr>
<tr>
<td>readdir</td>
<td>readdir_r</td>
</tr>
<tr>
<td>strtok</td>
<td>strtok_r</td>
</tr>
<tr>
<td>tmpnam</td>
<td>tmpnam_r</td>
</tr>
</tbody>
</table>
Use MT-Safe Routines

• 2) Safe
  – Global and static data are protected. Might not provide any concurrency between calls made by different threads.
  • Example: malloc in libc(3c)

• 3) MT-Safe
  – Safe and can provide a reasonable amount of concurrency.
Use MT-Safe Routines

• 4) Async-signal-safe
  – Can be safely called from a signal handler.
  – Example:
    • Not async-signal-safe: malloc(), pthread_getspecific()
    • Async-signal-safe: open(), read()
Problem 9: broken C++ idioms

- Singleton pattern
- Lazy initialization
- Reference counts
Why not drop down to threads?

• Shared data are a problem and we try to use locks

• Double Checking singleton pattern

Singleton & Singleton::Instance() {
    if (!pInstance_) // 1
    { pInstance_ = new Singleton; // 2 }
    return *pInstance_; // 3 }
Locking solution with threads

- While the thread assigns to `pInstance_`, all other stop in guard’s constructor.
- Each call to `Instance` incurs locking and unlocking the synchronization object.

```cpp
Singleton& Singleton::Instance() {
    // mutex_ is a mutex object
    Lock manages the mutex
    guard(mutex_);
    if (!pInstance_) {
        pInstance_ = new Singleton;
    }
    return *pInstance_;}
```
Double-checked locking pattern

Singleton& Singleton::Instance() {
    if (!pInstance_) // 1
    { // 2
        Guard myGuard(lock_); // 3
        if (!pInstance_) // 4
            { pInstance_ = new Singleton; } // 4
    }
    return *pInstance_; // 1
}
Example: Lazy Initialization

• The Sequential Version

```c
typedef struct
{
    int data1;
    int data2;
    ...
} A;

A *init_single_A()
{
    static A *single_A;
    if (single_A == NULL) {
        single_A = malloc(sizeof(A));
        single_A->data1 = ...;
    }
    return single_A;
}
```

It does not work in mt applications.
Example: Lazy Initialization Multithreaded Version

A *init_single_A()
{
    static A *single_A;
    lock();
    if (single_A == NULL) {
        single_A = malloc(sizeof(A));
        single_A->data1 = ...;
    }
    unlock();
    return single_A;
}

A *init_single_A()
{
    static A *single_A;
    A *temp = single_A;
    if (temp == NULL) {
        lock();
        if (single_A == NULL) {
            temp = malloc(sizeof(A));
            temp->data1 = ...;
        }
        single_A = temp;
    }
    unlock();
    return temp;
}

Not efficient

May be broken
Double-checked Locking

The compiler may reorder these two writes.

Even if the compiler does not reorder them, a thread on another processor may perceive the two writes in a different order.

Therefore, a thread on another processor may read wrong value of single_A->data1.
Fixing DCL

A possible fix.
Still broken on some architectures, e.g. PowerPC.

```c
A *init_single_A()
{
    static A *single_A;
    A *temp = single_A;
    if (temp == NULL) {
        lock();
        if (single_A == NULL) {
            temp = malloc(sizeof(A));
            temp->data1 = ...;
            ...
            memory_barrier();
            single_A = temp;
        }
        unlock();
    }
    return temp;
}
```
Memory Consistency Model

- **Pthreads**
  - No formal specification
  - Shared accesses must be synchronized by calling pthread synchronization functions.

- **C++/C**
  - C++03/C99: Assumes single thread program execution.
    - “volatile” restricts compiler optimization, but it does not address the memory consistency issue.
  - C++11/C11: Memory model for multithreaded C++ will be in C++11.
Memory Consistency Model

• OpenMP
  – Detailed clarification. No formal specification.
    • Each thread has a temporary view of shared memory.
    • A flush operation restricts the ordering of memory operations and synchronizes a thread's temporary view with shared memory. All threads must observe any two flush operations with overlapping variable lists in sequential order.
Memory Consistency Model

- Java: revised and clarified by JSR-133
  - Volatile variables
  - Final variables
  - Immutable objects (objects whose fields are only set in their constructor)
  - Thread-and memory-related JVM functionality and APIs such as class initialization, asynchronous exceptions, finalizers, thread interrupts, and the sleep, wait, and join methods of class Thread
C++11 solution

• Avoid writing codes that have deliberate data races. It is tricky and difficult to understand and debug.
• Use atomics on init variable
Agenda

- C11, C++11, C++14, SG5 TM goals and timelines
- C++ Standard Transactional Memory status
- Multithreading support in C11 and C++11
- The problems of Concurrency before C/C++ 11
- Language support
- Memory Model
- Fragen?
Can you do this today with TLS?

```cpp
extern std::string f(); // returns “Hello” from another TU
std::string foo(std::string const& s2) {
  __thread std::string s = f();
  s += s2;
  return s;
}
```
Non-atomic Variables

- Thread Local Storage (TLS)
- Static duration variables
  - Dynamic initialization
  - Destruction
Thread local storage variable

- Adopt existing practice
  ```
  __thread int a;
  ```
- Introduce new storage duration
  - Thread duration
  ```
  thread_local int var =3 ; //C++
  _Thread_local int var=5; //C
  ```
- Unique to each thread
- Accessible from every thread
- Address is not constant
Extend TLS

- Existing practice only supports static initialization and trivial destructors
  ```cpp
  std::string foo(std::string const& s2) {
    thread_local std::string s="hello";
    s+=s2;
    return s;
  }
  ```
- Want to extend it to dynamic initializers and destructors
  ```cpp
  thread_local vector<int> var=f();
  ```
- Dynamic initialization allows lazy init
- OS support may be needed
Initialization of static-duration variables

- Dynamic initialization is tricky
  - No syntax to order most initializations
- Without synchronization, potential data races
- With synchronization, potential deadlock
- Examine 2 kinds:
  - Function local statics
  - globals
Function-local static storage

- Initialization implicitly synchronized
  - While not holding any locks
- New algorithm contributed by Mike Burrows from Google

```c
void bar() {
    static my_class z(42+foo()); // initialization is thread-safe
    z.do_stuff();
}
```
Do not use a mutex during Initialization?

- Constructor declared as constexpr and satisfies the requirements for constant initialization
- Such objects are guaranteed to be init before any code is run as part of static init phase

```cpp
class my_class {
    int i;
    public:
        constexpr my_class():i(0){}
        my_class(int i_):i(i_){}
        void do_stuff();
};
my_class x; // static initialization with constexpr constructor
int foo();
my_class y(42+foo()); // dynamic initialization
void f()
{ y.do_stuff(); // is y initialized? }
```
Global variable

• Initialization implicitly synchronized
• Concurrent initialization enabled
• Initialization may not use a dynamically initialized object defined outside the translation unit

extern vector<int> e;
vector <int> u; //OK, default init
vector <int> v(u); //OK within this TU
vector <int> w(e); //undefined, outside of this TU
If you have to dynamically initialize...

- When `std::call_once` is used with an instance of `std::once_flag`, function is called exactly once

```cpp
my_class* p=0;
std::once_flag p_flag;
void create_instance() {
    p=new my_class(42+foo()); }
void baz() {
    std::call_once(p_flag,create_instance); p->do_stuff(); }
```
destruction

• First terminate all threads
• Execute destructors in a concurrent reverse of initialization
• Interleave namespace-scope vars with function-scope static vars
• Same restriction on use of vars outside current TU
Agenda

• C11, C++11, C++14, SG5 TM goals and timelines
• C++ Standard Transactional Memory status
• Multithreading support in C11 and C++11
• The problems of Concurrency before C/C++ 11
• Language support
• Memory Model
• Fragen?
Is this valid C++ today? Are these equivalent?

```cpp
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_acquire) != 1)
    // or: while (y.load() != 1)
    assert(x == 17);
```

```cpp
int x = 0;
atomic<int> y = 0;

Thread 1:
  x = 17;
y = 1;

Thread 2:
  while (y != 1)
    continue;
  assert(x == 17);
```
Memory Model

• One of the most important aspect of C++0x /C1x is almost invisible to most programmers
  – memory model
    • How threads interact through memory
    • What assumptions the compiler is allowed to make when generating code
    • 2 aspects
      – How things are laid out in memory
      – What happens when two threads access the same memory location and one of them is a modify
        » Data race
        » Modification order
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
    - $V=32$; \texttt{atomic\_store\_explicit}(&$a, 3, \text{memory\_order\_release}$);
  - Another thread acquires those writes
    - $i=$\texttt{atomic\_load\_explicit}(&$a, \text{memory\_order\_acquire}$);
      $i+v$;
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 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
• 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, atomicintptr_t, atomic_uintptr_t, atomic_size_t, atomic_ssize_t, atomicptrdiff_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, most 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=dependency-based ordering
Std::atomic<bool>

• Most basic std::atomic_bool, can be built from a non-atomic bool
• Can be constructed, initialized, 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
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<integral>

- this adds fetch_and, fetch_or, fetch_xor, and compound assignments like:
- +=, -=, &=, ^=, pre and post increment and decrement
- missing division, multiplication 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
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 copy-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
Atomic templates

• Makes an atomic type from a non-atomic type
  – Must be bitwise copyable and comparable
• Defined specializations for basic types and pointers
• Suggested specializations for alignment and size

  atomic <int *> aip = {0};
  aip=ip; aip+=4;
  atomic <small_type> ag={ .... };
  while (!ag.compare_swap(&ag, &g.g+4));
  atomic<circus > ac; //works, but not recommended
Atomic freedom

• Lock-free
  – Robust to crashes
  – Someone will make progress
  – C++14: “shall” obstruction-free and “should” for lock-free

• Wait-free
  – Operations completed in a bounded time
  – Cas vs ll/sc

• Address-free
  – Atomicity does not depend on using the same address
Lock-free atomics

• Large atomics have no hardware support
  – Implemented with locks
• Locks and signals don’t mix
  – Test for lock-free
• Compile-time macros for basic types
  – Always lock-free
  – Never lock-free
• RTTI for each type
Wait-free atomics

• Hard to implement without direct HW support
  – Resulting programs is usually HW-specific
  – Hard to be portable anyway

• IBM argued against it since ll/sc is not wait-free

• Few who write this seemed to cared anyway
  – No requirement for it
  – No query about it
Address-free atomics

- Memory may not have a consistent address
  - Processes sharing memory may not have the same address for that memory
  - Memory may be mapped into the address space twice
- Atomic operations must be address-free to work
  - One small tool for inter-process communication
- A lock-free operation is address-free
  - Not clear we can say this in a std way
  - But we will make our intent clear
Compiler Impact

- Memory model does not say how to make an application thread safe
  - Assumption is that source presented to compiler is thread safe
  - Undefined semantics for code with any data races
- Memory model describes legal transformations on already safe code
  - Compiler may not introduce any data races
- Memory model concerned with performance
  - Limited set of optimizations disallowed – may introduce data race
  - Allows some memory optimizations across locks
- Quality implementation
  - Most implementations already support a low quality implementation
  - Acquire/release operations seen as calls to opaque global functions – All shared variables may be referenced and modified
atomics

atomic<int> alp; alp=4; alp+=3;
atomic<int> current;
int desired, expected=current.load();
do desired=function(expected);
while(!current.compare_exchange_weak(expected, desired));
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
Memory Ordering Operations

class mem
{
    Memory_order_consumed,
    memory_order_relaxed,  // just atomic, no constraint
    memory_order_release,
    memory_order_acquire,
    memory_order_acq_rel,  // both acquire and release
    memory_order_seq_cst  // sequentially consistent
}; // (even stronger than cq_rel)

• 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
## Operations available on atomic types

<table>
<thead>
<tr>
<th></th>
<th>atomic_flag</th>
<th>bool/others</th>
<th>T*</th>
<th>integral</th>
</tr>
</thead>
<tbody>
<tr>
<td>test_and_set, clear</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>is_lock_free</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>load, store, exchange, compare_exchange_weak+strong</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>fetch_add (+=), fetch_sub (=), ++, --</td>
<td></td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>fetch_or (</td>
<td>=), fetch_and (&amp;=), fetch_xor (^=)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
Synchronizes with

- only between operations on atomic types
- operations on a data structure (locking a mutex) might provide this relationship if the data structure 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 ...

Thread 2
if (!x_init.ld_acq())
{
    lock();
    if (!x_init.ld...(())
        x = ...;
    x_init.store_rel(1);
    unlock();
}
... x ...
Sequential Consistency implies a total ordering

```cpp
std::atomic_bool x, y;
std::atomic_int z;
void write_x()
{
    x.store(true, std::memory_order_seq_cst); //2
}
void write_y()
{
    y.store(true, std::memory_order_seq_cst); //3
}
void read_x_then_y()
{
    while(!x.load(std::memory_order_seq_cst));
    if(y.load(std::memory_order_seq_cst)) //4
        ++z;
}
void read_y_then_x()
{
    while(!y.load(std::memory_order_seq_cst));
    if(x.load(std::memory_order_seq_cst)) //5
        ++z;
}

int main()
{
    x=false;
    y=false;
    z=0;
    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); //1
}
```
SC and happens-before

```cpp
std::atomic_bool x, y; std::atomic_int z; x = false; y = false; z = 0;

x.store(true); //2
while (!x.load());
if (y.load()) //4
{ ++z; }
while (!y.load());
if (x.load()) //5
{ ++z; }
assert(z.load() != 0); //1
```
Relaxed operations have little ordering requirement

```cpp
std::atomic_bool x, y;
std::atomic_int z;

void write_x_then_y() {
    x.store(true, std::memory_order_relaxed); //4
    y.store(true, std::memory_order_relaxed); //5
}

void read_y_then_x() {
    while(!y.load(std::memory_order_relaxed)); //3
    if(x.load(std::memory_order_relaxed) ) //2
        ++z;
}

int main() {
    x=false;
    y=false;
    z=0;
    std::thread a(write_x_then_y);
    std::thread b(read_y_then_x);
    a.join();
    b.join();
    assert(z.load()!=0); //1
}
```
std::atomic_bool x, y; std::atomic_int z; x = false; y = false; z = 0;

x.store(true); //4
y.store(true); //5
while(!y.load()); //3
If (x.load()); //2
{++z;}
assert(z.load() != 0); //1
Acquire-Release does not mean a total ordering

```c++
std::atomic_bool x, y;
std::atomic_int z;
void write_x()
{
    x.store(true, std::memory_order_release);
}
void write_y()
{
    y.store(true, std::memory_order_release);
}
void read_x_then_y()
{
    while (!x.load(std::memory_order_acquire));
    if (y.load(std::memory_order_acquire)) //3
        ++z;
}
void read_y_then_x()
{
    while (!y.load(std::memory_order_acquire));
    if (x.load(std::memory_order_acquire)) //2
        ++z;
}
int main()
{
    x=false;
    y=false;
    z=0;
    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); //1
}
```
std::atomic_bool x, y; std::atomic_int z; x=false; y=false; z=0;

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

• C11, C++11, C++14, SG5 TM goals and timelines
• C++ Standard Transactional Memory status
• Multithreading support in C11 and C++11
• The problems of Concurrency before C/C++ 11
• Language support
• Memory Model
• Fragen?
Food for thought and Q/A

• This is the chance to get a copy before you have to pay for it:
  – C: [http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf](http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf)

• Participate and feedback to Compiler
  – What features/libraries interest you or your customers?
  – What problem/annoyance you would like the Std to resolve?
  – Is Special Math important to you?
  – Do you expect 0x features to be used quickly by your customers?

• Talk to me at my blog:
My blogs and email address

• OpenMP CEO: http://openmp.org/wp/about-openmp/
  My Blogs: http://ibm.co/pCvPHR
  C++11 status: http://tinyurl.com/43y8xgf
  Boost test results http://www.ibm.com/support/docview.wss?rs=2239&context=SSJT9L&uid=swg27006911
  STM: https://sites.google.com/site/tmforcplusplus/

• Chair of WG21 SG5 Transactional Memory
• IBM and Canada C++ Standard Head of Delegation
• ISOCPP.org Director, Vice President http://isocpp.org/wiki/faq/wg21:michael-wong
• Vice Chair of Standards Council of Canada Programming Languages

• Tell us how you use OpenMP:
  • http://openmp.org/wp/whos-using-openmp/
Acknowledgement

• Some slides are based on committee presentations by various committee members, their proposals, and private communication
Hat Ihnen mein Vortrag gefallen?

Ich freue mich auf Ihr Feedback!