Concurrent programming in C++11

Multithreading is just one damn thing after, before, or simultaneous with another. --Andrei Alexandrescu

- Problems with C++98 memory model
- Double-checked locking pattern
- C++11 memory model
- Atomics
- Std::thread
- Mutex/Lock
- Conditional variable
- Future/Promise/Async

Problems with C++98

```
int X = 0;
int Y = 0;

// thread 1                          // thread 2
int r1 = X;                          int r2 = Y;
if ( 1 == r1 )                       if ( 1 == r2 )
    Y = 1;                               X = 1;

// can it be at the end of execution  r1 == r2 == 1 ?
```

Problems with C++98

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

```
// thread 1
x.a = 1;

// thread 1 may compiled:
struct s tmp = x;
tmp.a = 1;
x = tmp;
```

```
// thread 2
x.b = 1;

// thread 2 may be compiled:
struct s tmp = x;
tmp.b = 1;
x = tmp;
```

Singleton pattern

```
// in singleton.h:
class Singleton
{
public:
    static Singleton *instance();
    void other_method();
    // other methods ...
private:
    static Singleton *pinstance;
};

// in singleton.cpp:
Singleton *Singleton::pinstance = 0;

Singleton *Singleton::instance()
{
    if ( 0 == pinstance )
    {
        pinstance = new Singleton;  // lazy initialization
    }
    return pinstance;
}

// Usage:

   Singleton::istance()-> other_method();
```

Thread safe singleton construction

```
// in singleton.h:
class Singleton
{
public:
    static Singleton *instance();
    void other_method();
    // other methods ...
private:
    static Singleton *pinstance;
    static Mutex      lock_;
};

// in singleton.cpp:
Singleton *Singleton::pinstance = 0;

Singleton *Singleton::instance()
{
    Guard<Mutex> guard(lock_);  // constructor acquires lock_
    // this is now the critical section
    if ( 0 == pinstance )
    {
        pinstance = new Singleton;  // lazy initialization
    }
    return pinstance;
} // destructor releases lock_
```

Double checked locking pattern

```
Singleton *Singleton::instance()
{
    if ( 0 == pinstance )
    {
        Guard<Mutex> guard(lock_);  // constructor acquires lock_
        // this is now the critical section

        if ( 0 == pinstance )   // re-check pinstance
        {
            pinstance = new Singleton;  // lazy initialization
        }
    }                                   // destructor releases lock_
    return pinstance;
}



Singleton::istance()-> other_method();  // does not lock usually
```

Problems with DCLP

```
  if ( 0 == pinstance )
  {
        // ...
        pinstance = new Singleton;  // atomic?
        // ...
  }
  return pinstance;

// might use half-initialized pointer value
Singleton::istance()-> other_method();
```

- Pointer assignment may not be atomic
  - If can check an invalid, but not null pointer value

New expression

```
pinstance = new Singleton;  // how this is compiled?
```

- New expression include many steps
  - (1) Allocation space with ::operator new()
  - (2) Run of constructor
  - (3) Returning the pointer

- If the compiler does (1) + (3) and leaves (2) as the last step the pointer points to uninitialized memory area

Observable behavior in C++98

```
void foo()
{
    int x = 0, y = 0;         // (1)
    x = 5;                    // (2)
    y = 10;                   // (3)
    printf( "%d,%d", x, y); // (4)
}
```

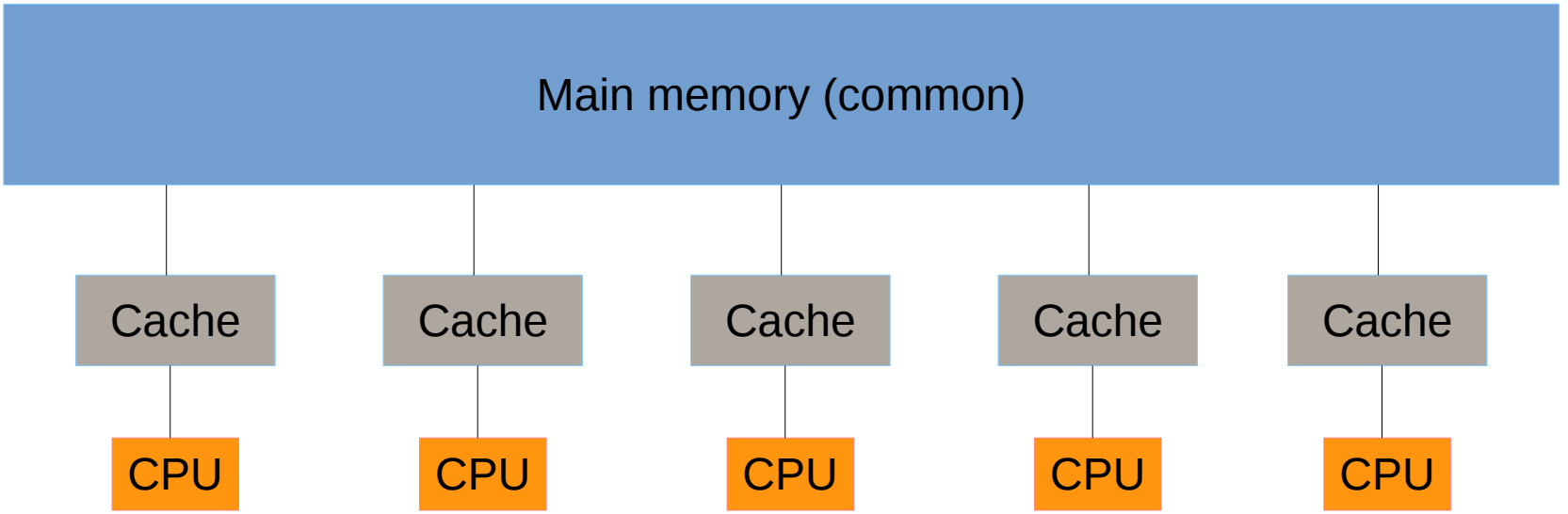
- What is visible for the outer word
  - I/O operations
  - Read/write volatile objects
- Defined by a singled-threaded mind

Sequence point

```
if ( 0 == pinstance )   // re-check pinstance
{
    // pinstance = new Singleton;
    Singleton *temp = operator new( sizeof(Singleton) );
    new (temp) Singleton;  // run the constructor
    pinstance = temp;
}
```

- The compiler can completely optimize out temp
- Even if we are using volatile temp we have issues

Modern hardware architecture



Singleton pattern

```
Singleton *Singleton::instance()
{
    Singleton *temp = pInstance;    // read pInstance

    Acquire();    // prevent visibility of later memory operations
                  // from moving up from this point

    if ( 0 == temp )
    {
        Guard<Mutex> guard(lock_);
        // this is now the critical section

        if ( 0 == pinstance )   // re-check pinstance
        {
            temp = new Singleton;

            Release();  // prevent visibility of earlier memory operations
                        // from moving down from this point

            pinstance = temp;   // write pInstance
        }
    }
    return pinstance;
}
```

C++11 memory model

```
int x, y;

// thread 1       |       // thread 2
x = 1;            |       cout << y << ", ";
y = 2;            |       cout << x << endl;
```

In C++03 not even Undefined Behavior

In C++11 Undefined Behavior

C++11 memory model

```
std::atomic<int> x, y;

// thread 1       |       // thread 2
x.store(1);       |       cout << y.load() << ", ";
y.store(2);       |       cout << x.load() << endl;


Equivalent to:

int x, y;
mutex x_mutex, y_mutex;

// thread 1       |       // thread 2
x_mutex.lock()    |       y_mutex.lock();
x = 1;            |       cout << y << ", ";
x_mutex.unlock()  |       y_mutex.unlock();
y_mutex.lock()    |       x_mutex.lock();
y = 2;            |       cout << x << endl;
y_mutex.unlock()  |       x_mutex.unlock();
```

C++11 memory model (default)

```
std::atomic<int> x, y;
x.store(0); y.store(0);

// thread 1      |       // thread 2
x.store(1);      |       cout << y.load() << ", ";
y.store(2);      |       cout << x.load() << endl;


Result can be:


0 0
2 1
0 1
// never prints: 2 0
```

Sequential consistency: atomics == atomic load/store + ordering

Terminology

- Only minimal progress guaranties are given:
  - unblocked threads will make progress
  - implementation should ensure that writes in a thread should be visible in other threads "in a finite amount of time".
- The A happens before B relationship:
  - A is sequenced before B or
  - A inter-thread happens before B

    == there is a synchronization point between A and B
- Synchronization point:
  - thread creation sync with start of thread execution
  - thread completion sync with the return of join()
  - unlocking a mutex sync with the next locking of that mutex

Terminology

- Memory location
  - an object of scalar type
  - a maximal sequence of adjacent bit-fields all having non-zero width
- Data race

  A program contains **data race** if contains two actions in different threads, at least one is not "atomic" **and** neither happens before the other.
- Two threads of execution can update and access separate memory locations without interfering each others

Sequential consistent

- Sequential consistent (default behavior)
  - Leslie Lamport, 1979
  - Each thread is sequential,operations can not be reordered
  - The operations of each thread appear in this sequence
- C++ memory model contract
  - Programmer ensures that the program has no data race
  - System guarantees sequentially consistent execution

Memory ordering

- memory_order_seq_cst (default)
- memory_order_consume
- memory_order_acquire
- memory_order_release
- memory_order_acq_rel
- memory_order_relaxed

X86/x86_64 does not require additional instructions to implement acquire-release ordering

Relaxed memory order

- Each memory location has a total modification order
  - But this may be not observable directly
- Memory operations performed by
  - The same thread and
  - On the same memory location

  are not reordered with respect of modification order

C++11 memory model (others)

```
std::atomic<int> x, y;

// relaxed
// thread 1                          |  // thread 2
x.store(1, memory_order_relaxed); |  cout << y.load(memory_order_relaxed) << ", ";
y.store(2, memory_order_relaxed); |  cout << x.load(memory_order_relaxed) << endl;

// Defined, atomic, but not ordered, result may be:
0 0
2 1
0 1
2 0

// acquire-release
// thread 1                          |  // thread 2
x.store(1, memory_order_release); |  cout << y.load(memory_order_acquire) << ", ";
y.store(2, memory_order_release); |  cout << x.load(memory_order_acquire) << endl;

// In C++11 Defined and the result can be:
0 0
2 1
0 1
// never prints: 2 0, but can be faster than strict ordering.
// results may be different in more complex programs
```

std::thread

```
class thread
{
public:
  typedef native_handle ...;
  typedef id ...;

  thread() noexept;                 // does not represent a thread
  thread( thread&& other) noexept; // move constructor
  ~thread();                        // if joinable() calls std::terminate()

  template <typename Function, typename... Args> // copies args to thread local
  explicit thread( Function&& f, Arg&&... args); // then execute f with args

  thread(const thread&) = delete;  // no copy
  thread& operator=(thread&& other) noexept;  // move
  void swap( thread& other);  // swap

  bool joinable() const; // thread object owns a physical thread
  void join();           // blocks current thread until *this finish
  void detach();         // separates physical thread from the thread object

  std::thread::id get_id() const;              // std::this_thread
  static unsigned int hardware_concurrency();  // supported concurrent threads
  native_handle_type native_handle();          // e.g. thread id
};
```

Usage of std::thread

```
void f( int i, const std::string&);
{
    std::cout << "Hello concurrent world" << std::endl;
}

int main()
{
    int i = 3;
    std::string s("Hello");

    // Will copy both i and s
    // We can prevent the copy by using reference wrapper
    // std::thread t( f, std::ref(i), std::ref(s));
    std::thread t( f, i, s);

    // if the thread destructor runs and the thread is joinable, than
    // std::system_error will be thrown.
    // Use join() or detach() to avoid that.
    t.join();

    return 0;
}
```

Issue with join()

- If the thread destructor called when the thread is still *joinable* std::system_error will be thrown
- Alternatives are not really feasible:
- Implicit join:
  - The destructor waits until the thread execution is completed
  - Hard-to detect performance issues
- Implicit detach
  - The destructor may run, but the underlying thread is still under execution
  - We may destroy resources still used by the thread
- Scoped_thread or thread_strategy parameters

Scoped thread

```
class scoped_thread  // Anthony Williams
{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_): t(std::move(t_))
    {
        if(!t.joinable())
            throw std::logic_error(“No thread”);
    }
    ~scoped_thread()
    {
        t.join();
    }
    scoped_thread(scoped_thread const&)=delete;
    scoped_thread& operator=(scoped_thread const&)=delete;
};

struct func;

void f()
{
    int some_local_state;
    scoped_thread t(std::thread(func(some_local_state)));
    do_something_in_current_thread();
}
```

Usage of std::thread

```
struct func
{
    int& i;
    func(int& i_) : i (i_) { }

    void operator()()
    {
        for(unsigned int j=0; j < 1000000; ++j)
        {
            do_something(i);  // i refers to a destroyed variable
        }
    }
};

void oops()
{
    int some_local_state=0;

    func my_func(some_local_state);

    std::thread my_thread(my_func);

    my_thread.detach();  // don't wait the thread to finish
}  // i is destroyed, but the thread is likely still running..
```

std::thread works with containers

```
void do_work(unsigned id);

void f()
{
    std::vector<std::thread> threads;
    for(unsigned i=0;i<20;++i)
    {
        threads.push_back(std::thread(do_work,i));
    }
    std::for_each(threads.begin(),threads.end(),
                  std::mem_fn(&std::thread::join));
}
```

std::thread works with containers

```
// std::thread::id identifiers returned by std::this_thread::get_id()
// it returns std::thread::id() if there is no associated thread.
std::thread::id master_thread;
void some_core_part_of_algorithm()
{
    if(std::this_thread::get_id()==master_thread)
    {
        do_master_thread_work();
    }
    do_common_work();
}


// gives a hint about the available cores. Be aware of
// “oversubscription", i.e. using more threads than cores we have.
std::thread::hardware_concurency()
```

Syncronization objects: mutex

```
#include <mutex>

void f()
{
    std::mutex m;
    int sh;  // shared data
    // ...
    m.lock();
    // manipulate shared data:
    sh+=1;
    m.unlock();
}


// Recursive mutex
std::recursive_mutex m;
int sh;  // shared data
// ...
void f(int i)
{
// ...
    m.lock();
    // manipulate shared data:
    sh+=1;
    if (--i>0) f(i);
    m.unlock();
    // ...
}
```

Syncronization objects: timed mutex

```
void f1()
{
    std::timed_mutex m;
    int sh;  // shared data
    // ...
    if (m.try_lock_for(std::chrono::seconds(10)))
    {
      // manipulate shared data:
      sh+=1;
      m.unlock();
    }
    else
      // we didn't get the mutex; do something else
}
void f2()
{
    std::timed_mutex m;
    int sh;  // shared data
    // ...
    if (m.try_lock_until(midnight))
    {
      // manipulate shared data:
      sh+=1;
      m.unlock();
    }
    else
      // we didn't get the mutex; do something else
}
```

RAII support

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> l;
std::mutex     m;

void add_to_list(int value);
{
    // lock acquired - with RAII style lock management
    std::lock_guard< std::mutex > guard(m);
    l.push_back(value);
}   // lock released
```

Pointers or references pointing out from the guarded area may be an issue!

Can this go dead-locked?

```
bool operator<( T const& lhs, T const& rhs)
{
    if ( &lhs == &rhs )
   return false;

    std::lock_guard< std::mutex > guard(lhs.m)
    std::lock_guard< std::mutex > guard(rhs.m)

    return lhs.data < rhs.data;
}



// thread1          |          thread2

    a < b          |           b < a
```

Correct solution

```
bool operator<( T const& lhs, T const& rhs)
{
    if ( &lhs == &rhs )
        return false;

    // std::lock - lock two or more mutexes
    std::lock( lhs.m, rhs.m);
    std::lock_guard< std::mutex >  lock_lhs( lhs.m, std::adopt_lock);
    std::lock_guard< std::mutex >  lock_rhs( rhs.m, std::adopt_lock);

    return lhs.data < rhs.data;
}
```

Unique_lock with defer_lock

```
bool operator<( T const& lhs, T const& rhs)
{
    if ( &lhs == &rhs )
        return false;

    // std::unique_locks constructed with defer_lock can be locked
    // manually, by using lock() on the lock object ...
    std::unique_lock< std::mutex >  lock_lhs( lhs.m, std::defer_lock);
    std::unique_lock< std::mutex >  lock_rhs( rhs.m, std::defer_lock);
    // lock_lhs.owns_lock() now false

    // ... or passing to std::lock
    std::lock( lock_lhs, lock_rhs);  // designed to avoid dead-lock
    // also there is an unlock() memberfunction

    // lock_lhs.owns_lock() now true
    return lhs.data < rhs.data;
}
```

Unique_lock only moveable

```
std::unique_lock<std::mutex> get_lock()
{
    extern std::mutex some_mutex;
    std::unique_lock<std::mutex> lk(some_mutex);
    prepare_data();
    return lk; // same as std::move(lk),
               // return does not require std::move
}

void process_data()
{
    std::unique_lock<std::mutex> lk(get_lock());
    do_something();
}
```

Concurrent singleton

```
template <typename T>
class MySingleton
{
public:
    std::shared_ptr<T> instance()
    {
        std::call_once( resource_init_flag, init_resource);
        return resource_ptr;
    }
private:
    void init_resource()
    {
        resource_ptr.reset( new T(...) );
    }
    std::shared_ptr<T> resource_ptr;
    std::once_flag     resource_init_flag; // can't be moved or copied
};
```

Meyers singleton

```
// Meyers singleton:
// C++11 guaranties: local static is initialized in a thread safe way
//
class MySingleton;
MySingleton& MySingletonInstance()
{
    static MySingleton _instance;
    return _instance;
}
```

Spin lock

```
bool flag;   // waiting for this flag
std::mutex m;

void wait_for_flag()
{
    std::unique_lock<std::mutex> lk(m);
    while(!flag)
    {
        lk.unlock();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
        lk.lock();
    }
}
```

Condition variable

```
std::mutex                my_mutex;
std::queue< data_t >      my_queue;
std::conditional_variable data_cond;  // conditional variable

void producer()
{
    while ( more_data_to_produce() )
    {
        const data_t data = produce_data();
        std::lock_guard< std::mutex > prod_lock(my_mutex);  // guard the push
        my_queue.push(data);
        data_cond.notify_one(); // notify the waiting thread to evaluate cond.
    }
}

void consumer()
{
    while ( true )
    {
        std::unique_lock< std::mutex > cons_lock(my_mutex); // not lock_guard
        data_cond.wait(cons_lock,                    // returns if lamdba returns true
                    [&my_queue]{return !my_queue.empty();}); // else unlocks and waits
        data_t data = my_queue.front();  // lock is hold here to protect pop...
        my_queue.pop();
        cons_lock.unlock();              // ... until here
        consume_data(data);
    }
}
```

Condition variable

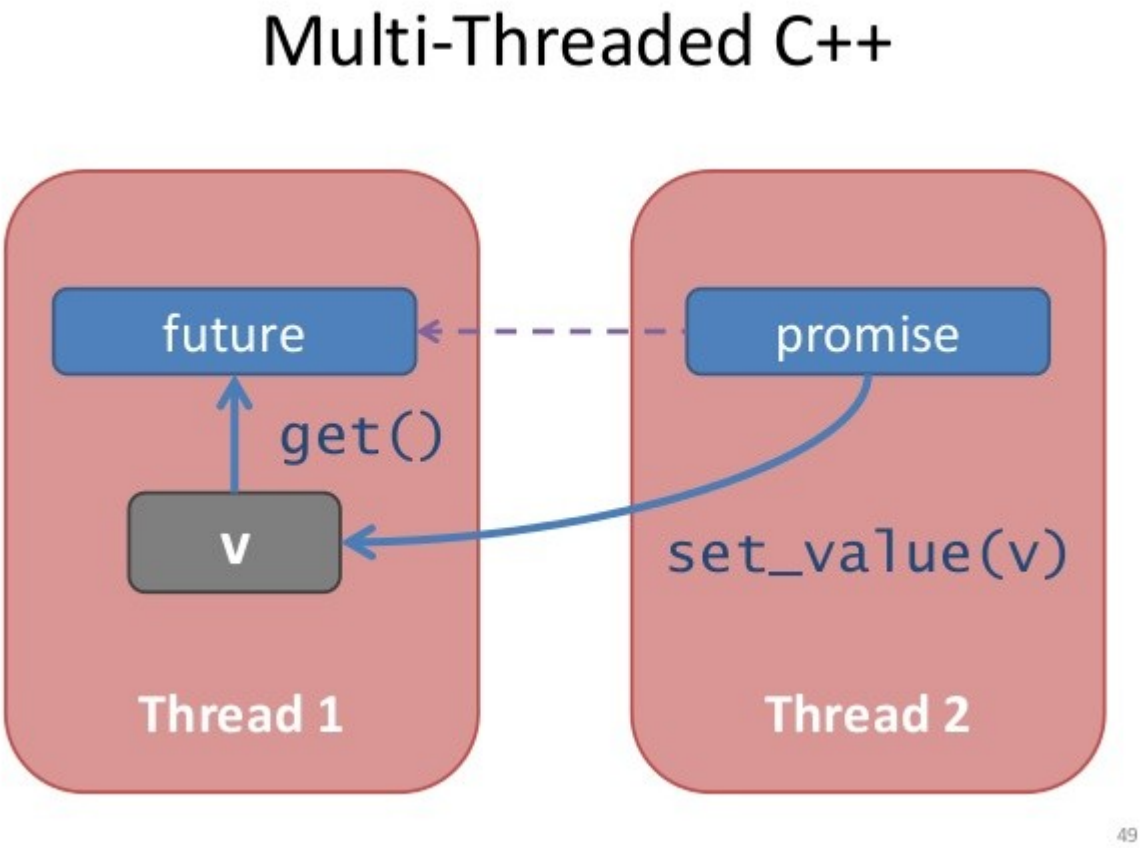
- During the wait the condition variable may check the condition any time
- But under the protection of the mutex and returns immediately if condition is true.

- Spurious wake: wake up without notification from other thread. Undefined times and frequency -> better to avoid functions with side effect (e.g.using a counter in lambda to check how many notifications were is bad)

Future

- 1976 Daniel P. Friedman and David Wise: promise
- 1977 Henry Baker and Carl Hewitt: future
- Future: a read-only placeholder view of a variable or exception
- Promise: a writeable, single assignment container (to set the future)
- Communication channel: promise → future
- std::future the
  - Only instance to refer the async event
  - Move-only
- std::shared_future
  - Multiple instances referring to the same event
  - Copiable
  - All instances will be ready on the same time

Future-Promise



std::async

```
#include <future>
#include <iostream>

int f(int);
void do_other_stuff();

int main()
{
    std::future<int> the_answer = std::async(f,1);
    do_other_stuff();
    std::cout<< "The answer is " << the_answer.get() << std::endl;
}

// The std::async() executes the task either in a new thread or on get()

// starts in a new thread
auto fut1 = std::async(std::launch::async, f, 1);
// run in the same thread on wait() or get()
auto fut2 = std::async(std::launch::deferred, f, 2);
// default: implementation chooses
auto fut3 = std::async(std::launch::deferred | std::launch::async, f, 3);
// default: implementation chooses
auto fut4 = std::async(f, 4);

// If no wait() or get() is called, then the task may not be executed at all.
```

Exceptions

```
double square_root(double x)
{
    if ( x < 0 )
    {
        throw std::out_of_range("x<0");
    }
    return sqrt(x);
}

int main()
{
    std::future<double> fut = std::async( square_root, -1);
    // do something else...
    double res = fut.get(); // f becomes ready on exception and rethrows
}                           // exception object could be a copy of original
```

Exceptions

```
void asyncFun( std::promise<int> myPromise)
{
    int result;
    try
    {
        // calculate the result
        myPromise.set_value(result);
    }
    catch ( ... )
    {
        myPromise.set_exception(std::current_exception());
    }
}

// In the calling thread:
int main()
{
    std::promise<int> intPromise;
    std::future<int> intFuture = intPromise.getFuture();
    std::thread t(asyncFun, std::move(intPromise));

    // do other stuff here, while asyncFun is working

    int result = intFuture.get();  // may throw MyException
    return 0;
}
```

Exceptions

```
// Example from Stroustrup

template<class T, class V>
struct Accum     // simple accumulator function object
{
    T* b;
    T* e;
    V val;
    Accum(T* bb, T* ee, const V& vv) : b{bb}, e{ee}, val{vv} {}
    V operator() () { return std::accumulate(b,e,val); }
};

void comp(vector<double>& v)   // spawn many tasks if v is large enough
{
    if (v.size()<10000) return std::accumulate(v.begin(),v.end(),0.0);

    auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
    auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
    auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
    auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

    return f0.get()+f1.get()+f2.get()+f3.get();
}
```

C++17

- resumable functions
  - async ... wait
- transactional memory
- continuation
  - then()
  - when_any()
  - when_all()
- parallel STL (Intel TBB)?

- Critics on C++ concurrency:

  Bartosz Milewski's blog: Broken promises - C++0x futures

  http://bartoszmilewski.com/2009/03/03/broken-promises-c0x-futures/

  MeetingC++ - Hartmut Kaiser: Plain Threads are the GOTO of todays computing

  https://www.youtube.com/watch?v=4OCUEgSNIAY