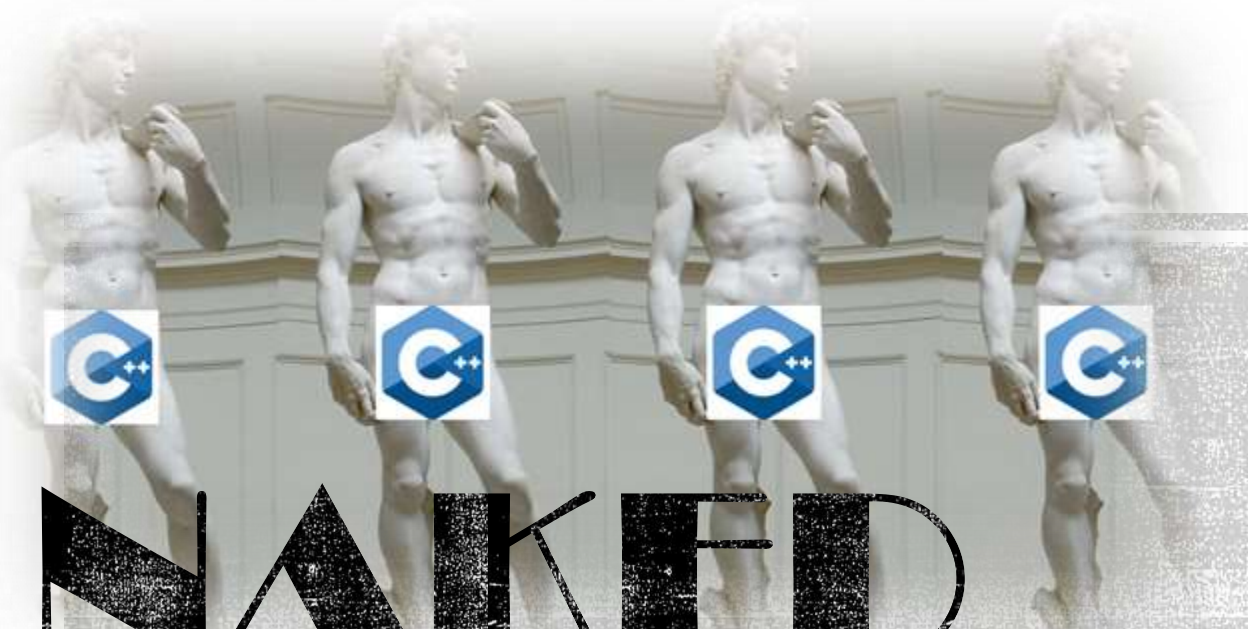


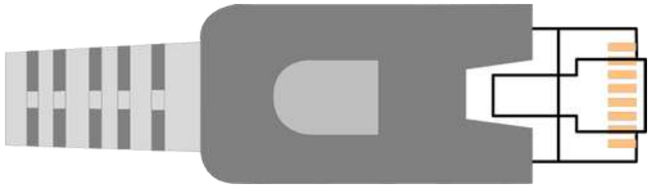
NAKED COROUTINES

LIVE

CppCon 2017

live with Networking

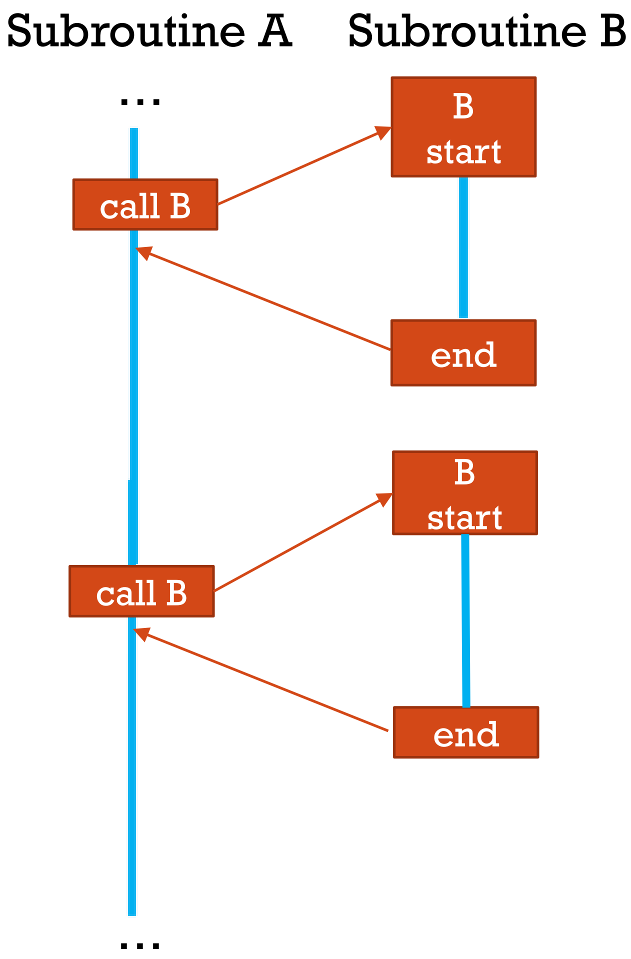
Gor Nishanov • Visual C++ Team • Microsoft



DESIGN PRINCIPLES

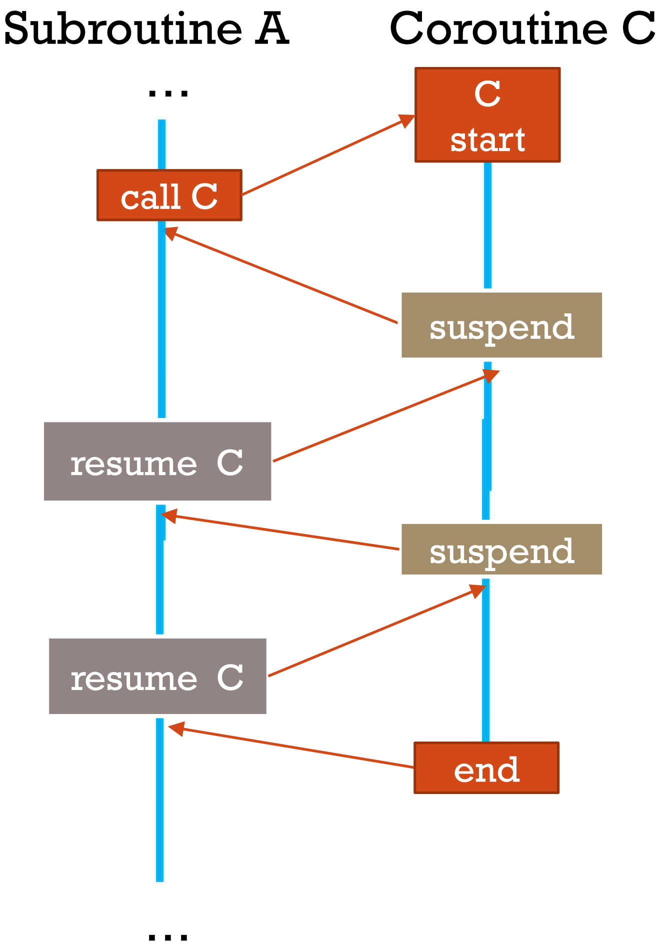
- **Scalable** (to **b**illions of concurrent coroutines)
- **Efficient** (resume and suspend operations comparable in cost to a function call overhead)
- Seamless interaction with existing facilities **with no overhead**
- **Open ended** coroutine machinery allowing library designers to develop coroutine libraries exposing various high-level semantics, such as generators, tasks, async streams and more.
- **Usable** in environments where **exceptions** are forbidden or **not available**

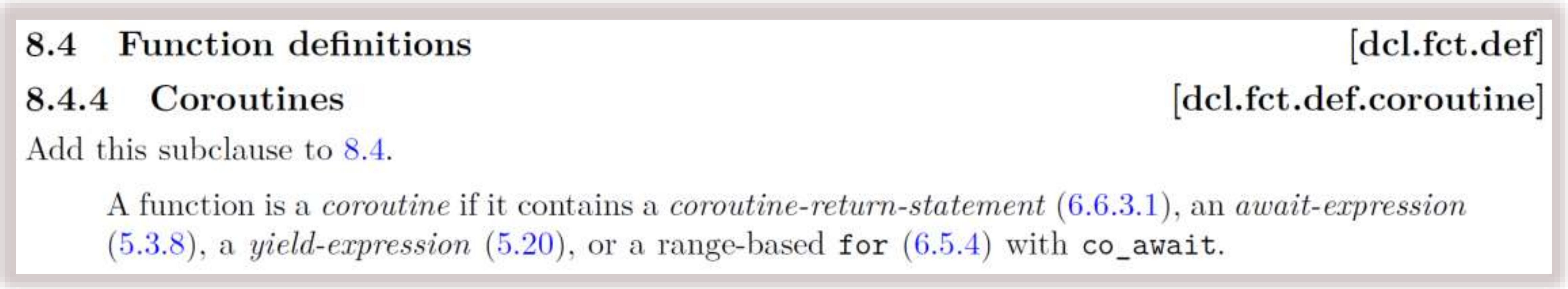
COROUTINES



- Introduced in 1958 by Melvin Conway
- Donald Knuth, 1968: "generalization of subroutine"

	subroutines	coroutines
call	Allocate frame, pass parameters	Allocate frame, pass parameters
return	Free frame, return result	Free frame, return eventual result
suspend	x	yes
resume	x	yes





8.4 Function definitions [dcl.fct.def]

8.4.4 Coroutines [dcl.fct.def.coroutine]

Add this subclause to 8.4.

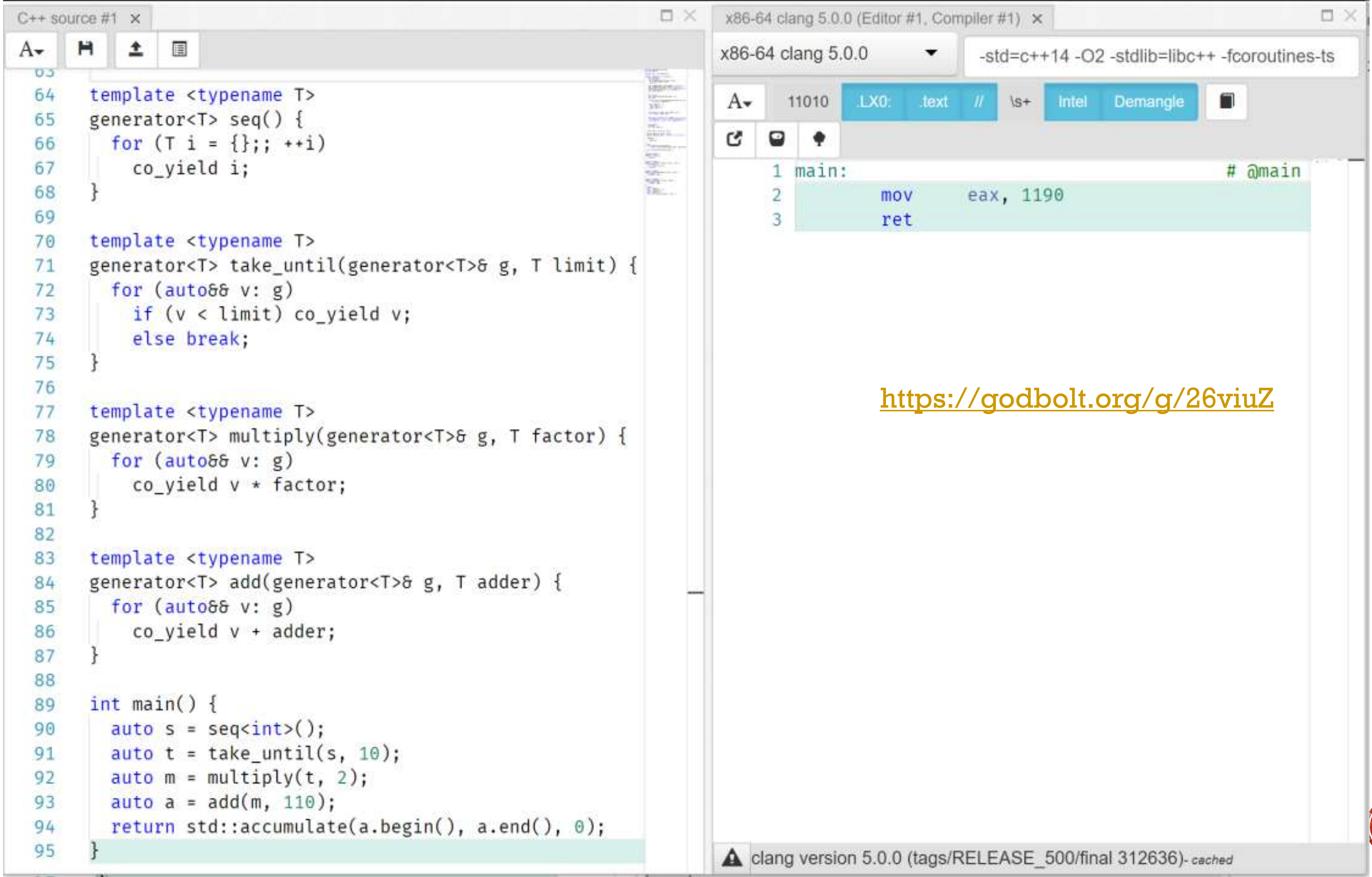
A function is a *coroutine* if it contains a *coroutine-return-statement* (6.6.3.1), an *await-expression* (5.3.8), a *yield-expression* (5.20), or a range-based `for` (6.5.4) with `co_await`.

```
generator<char> hello() {
   for (char ch: "Hello, world\n")
      co_yield ch;
}

int main() {
   for (char ch : hello())
      cout << ch;
}
```

```
future<void> sleepy() {
   cout << “Going to sleep…\n";
   co_await sleep_for(1ms);
   cout << “Woke up\n";
   co_return 42;
}

int main() {
   cout << sleepy.get();
}
```



```cpp
template <typename T>
generator<T> seq() {
  for (T i = {};; ++i)
    co_yield i;
}

template <typename T>
generator<T> take_until(generator<T>& g, T limit) {
  for (auto&& v: g)
    if (v < limit) co_yield v;
    else break;
}

template <typename T>
generator<T> multiply(generator<T>& g, T factor) {
  for (auto&& v: g)
    co_yield v * factor;
}

template <typename T>
generator<T> add(generator<T>& g, T adder) {
  for (auto&& v: g)
    co_yield v + adder;
}

int main() {
  auto s = seq<int>();
  auto t = take_until(s, 10);
  auto m = multiply(t, 2);
  auto a = add(m, 110);
  return std::accumulate(a.begin(), a.end(), 0);
}
```

x86-64 clang 5.0.0

-std=c++14 -O2 -stdlib=libc++ -fcoroutines-ts

```asm
main:                                   # @main
        mov     eax, 1190
        ret
```

https://godbolt.org/g/26viuZ

⚠ clang version 5.0.0 (tags/RELEASE_500/final 312636)- *cached*

GIFTS FROM TORONTO 2017

Coroutine TS

Networking TS

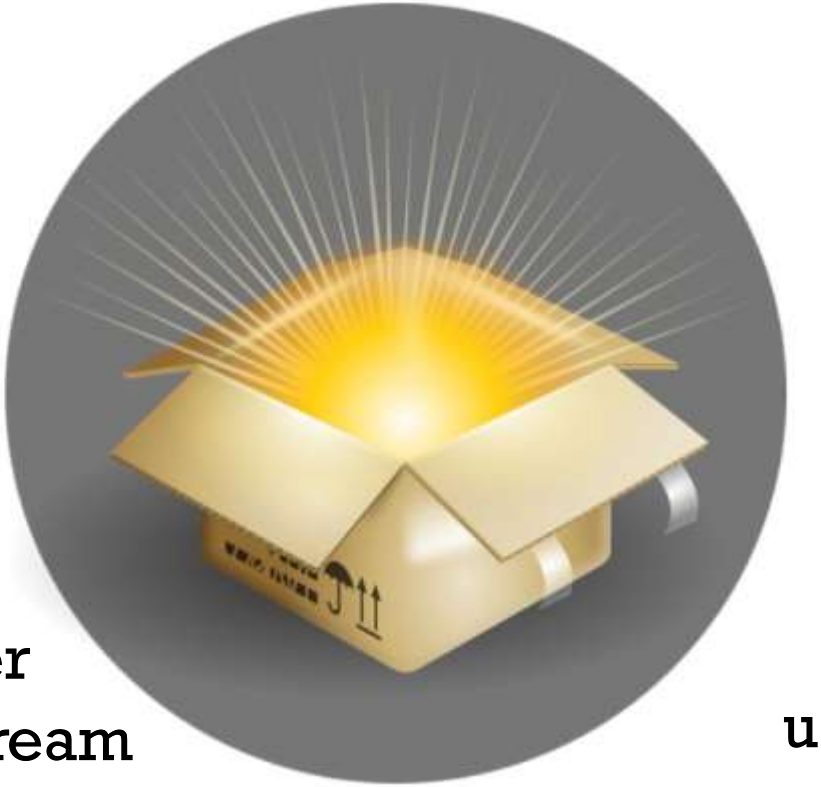
OPENING THE NETWORKING TS BOX!

io_context

+ and more nifty things

executors

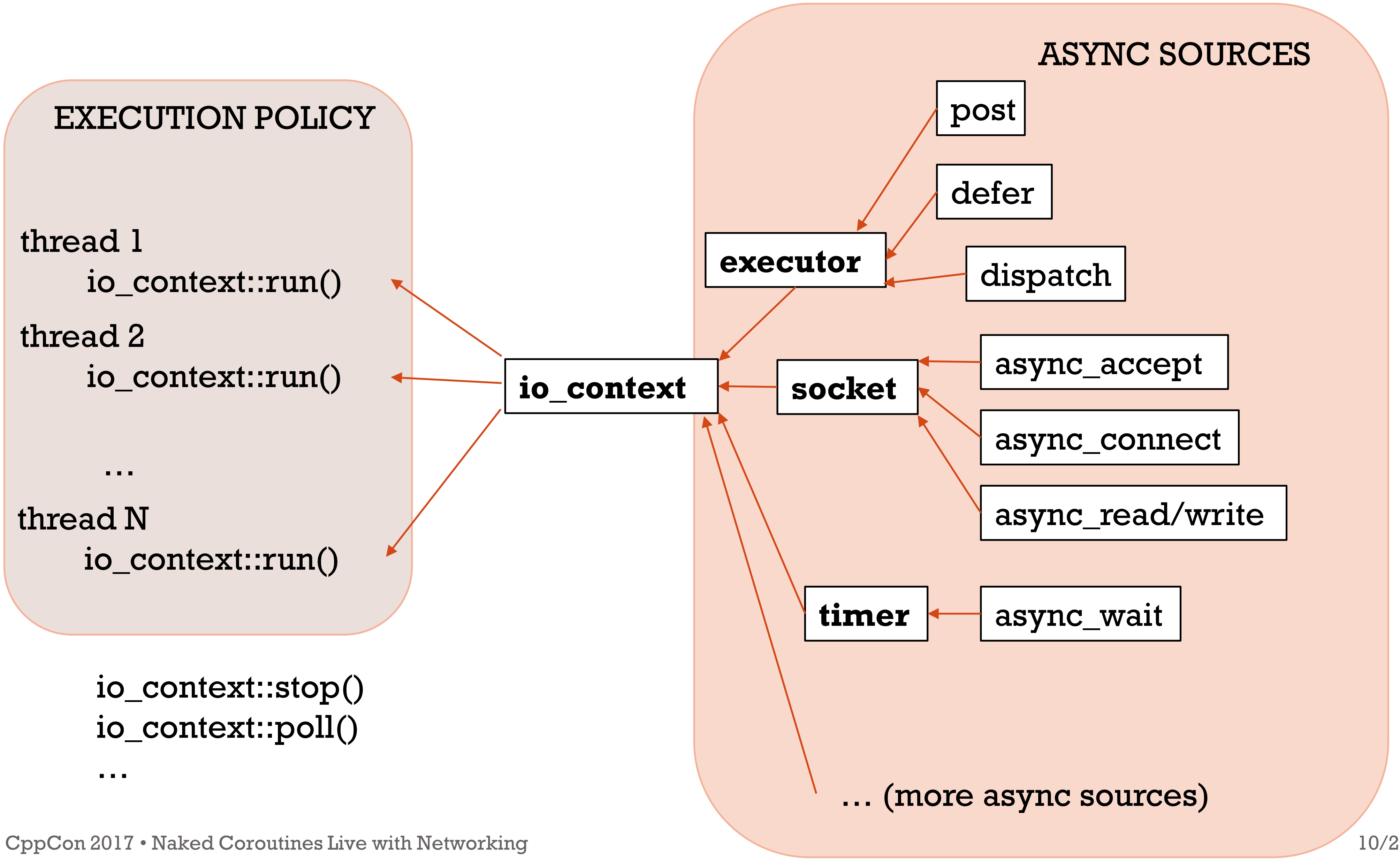
timers



tcp::endpoint
tcp::socket
tcp::acceptor
tcp::resolver
tcp::iostream

udp::endpoint
udp::socket
udp::resolver

NETWORKING TS – IO_CONTEXT



EXECUTION POLICY

thread 1
io_context::run()

thread 2
io_context::run()

…

thread N
io_context::run()

io_context

ASYNC SOURCES

executor

post

defer

dispatch

socket

async_accept

async_connect

async_read/write

timer

async_wait

… (more async sources)

io_context::stop()
io_context::poll()
…

SIMPLE TIMER EXAMPLE

```
int main() {
  io_context io;

  system_timer slow_timer(io, hours(15));
  slow_timer.async_wait([](auto) {
    puts("Timer fired");
  });

  system_timer fast_timer(io, seconds(1));
  fast_timer.async_wait([&io](auto) {
    io.stop();
  });

  io.run();
}
```

BEAUTIFUL TCP SERVER

```
struct session {
  session(net::io_context &ioc, net::ip::tcp::socket s, size_t block_size)
      : io_context_(ioc), socket_(std::move(s)), block_size_(block_size),
        buf_(block_size), read_data_length_(0)
  {}

  void start() {
    std::error_code set_option_err;
    net::ip::tcp::no_delay no_delay(true);
    socket_.set_option(no_delay, set_option_err);
    if (!set_option_err) {
      socket_.async_read_some( net::buffer(buf_.data(), block_size_),
        make_custom_alloc_handler( allocator_,
          [this](auto ec, auto n) { handle_read(ec, n); }));
      return;
    }

    net::post(io_context_, [this] { destroy(this); });
  }

  void handle_read(const std::error_code &err, size_t length) {
    if (!err) {
      read_data_length_ = length;
      async_write(socket_, net::buffer(buf_.data(), read_data_length_),
        make_custom_alloc_handler( allocator_,
          [this](auto ec, auto) { handle_write(ec); }));
      return;
    }

    net::post(io_context_, [this] { destroy(this); });
  }
```

```
void handle_write(const std::error_code &err) {
    if (!err) {
      socket_.async_read_some(net::buffer(buf_.data(), block_size_),
        make_custom_alloc_handler( allocator_,
          [this](auto ec, auto n) { handle_read(ec, n); }));
      return;
    }

    net::post(io_context_, [this] { destroy(this); });
  }

  static void destroy(session *s) { delete s; }

private:
  net::io_context &io_context_;
  net::ip::tcp::socket socket_;
  size_t block_size_;
  std::vector<char> buf_;
  size_t read_data_length_;
  handler_allocator allocator_;
};
```

```
struct server {
  server(net::io_context &ioc, const net::ip::tcp::endpoint &endpoint,
         size_t block_size)
      : io_context_(ioc), acceptor_(ioc, endpoint), block_size_(block_size)
  {
    acceptor_.listen();

    start_accept();
  }

  void start_accept()
  {
    acceptor_.async_accept(
        [this](auto ec, auto s) { handle_accept(ec, std::move(s)); });
  }

  void handle_accept(std::error_code err, net::ip::tcp::socket s)
  {
    if (!err) {
      session *new_session = new session(io_context_, std::move(s), block_size_);
      new_session->start();
    }
    start_accept();
  }

private:
  net::io_context &io_context_;
  net::ip::tcp::acceptor acceptor_;
  size_t block_size_;
};
```

UNBOXING THE COROUTINES

and that is all you get!

suspend_never

suspend_always

coroutine_handle

coroutine_traits

co_await

co_yield

co_return

READ THE MANUAL?

© ISO 2017 — All rights reserved

ISO/IEC JTC1 SC22 WG21 D4680
Date: 2017-07-30
ISO/IEC TS 22277
ISO/IEC JTC1 SC22
Secretariat: ANSI

Programming Languages — C++ Extensions for Coroutines

Langages de programmation — Extensions C++ pour les Coroutines

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Proposed Draft Technical Specification
Document stage: (20) Committee
Document Language: E

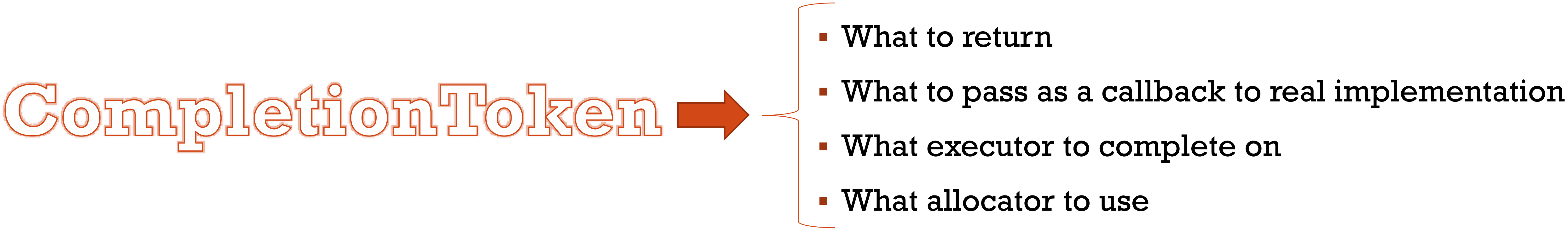
LIVE

THE EASY WAY

ASYNC INITIATING FUNCTION

```
template <typename BufferSequence, typename CompletionToken>
auto async_xyz(BufferSequence const& buffers, CompletionToken handler)
{
  async_completion<CompletionToken, void(std::error_code, std::size_t)> init(handler);

  impl.real_async_xyz(buffers, init.completion_handler);
  return init.result.get();
}
```



CompletionToken

- What to return
- What to pass as a callback to real implementation
- What executor to complete on
- What allocator to use

TRAIT SPECIALIZATION FOR USE_BOOST_FUTURE

```
template <>
struct async_result<use_boost_future_t,  void(std::error_code, size_t)> {
  using return_type = boost::future<size_t>;
  struct completion_handler_type {
     boost::promise<size_t> p;
     completion_handler_type(use_boost_future_t const&) {}
     void operator() (std::error_code const& ec, size_t n) {
       if (ec) p.set_exception(std::system_error(ec));
       else p.set_value(n);
    }
  };
  explicit async_result(completion_handler_type &h) : fut(h.p.get_future()) {}
  auto get() { return std::move(fut); }
private:
  boost::future<size_t> fut;
};
```

LIVE

BEAUTIFUL TCP SERVER

```
struct session {
  session(net::io_context &ioc, net::ip::tcp::socket s, size_t block_size)
      : io_context_(ioc), socket_(std::move(s)), block_size_(block_size),
        buf_(block_size), read_data_length_(0)
  {}

  void start() {
    std::error_code set_option_err;
    net::ip::tcp::no_delay no_delay(true);
    socket_.set_option(no_delay, set_option_err);
    if (!set_option_err) {
      socket_.async_read_some( net::buffer(buf_.data(), block_size_),
        make_custom_alloc_handler( allocator_,
          [this](auto ec, auto n) { handle_read(ec, n); }));
      return;
    }

    net::post(io_context_, [this] { destroy(this); });
  }

  void handle_read(const std::error_code &err, size_t length) {
    if (!err) {
      read_data_length_ = length;
      async_write(socket_, net::buffer(buf_.data(), read_data_length_),
        make_custom_alloc_handler( allocator_,
          [this](auto ec, auto) { handle_write(ec); }));
      return;
    }

    net::post(io_context_, [this] { destroy(this); });
  }
void handle_write(const std::error_code &err) {
    if (!err) {
      socket_.async_read_some(net::buffer(buf_.data(), block_size_),
        make_custom_alloc_handler( allocator_,
          [this](auto ec, auto n) { handle_read(ec, n); }));
      return;
    }

    net::post(io_context_, [this] { destroy(this); });
  }

  static void destroy(session *s) { delete s; }

private:
  net::io_context &io_context_;
  net::ip::tcp::socket socket_;
  size_t block_size_;
  std::vector<char> buf_;
  size_t read_data_length_;
  handler_allocator allocator_;
};
```

```
struct server {
  server(net::io_context &ioc, const net::ip::tcp::endpoint &endpoint,
         size_t block_size)
      : io_context_(ioc), acceptor_(ioc, endpoint), block_size_(block_size)
  {
    acceptor_.listen();

    start_accept();
  }

  void start_accept()
  {
    acceptor_.async_accept(
        [this](auto ec, auto s) { handle_accept(ec, std::move(s)); });
  }

  void handle_accept(std::error_code err, net::ip::tcp::socket s)
  {
    if (!err) {
      session *new_session = new session(io_context_, std::move(s), block_size_);
      new_session->start();
    }
    start_accept();
  }

private:
  net::io_context &io_context_;
  net::ip::tcp::acceptor acceptor_;
  size_t block_size_;
};
```

SAME BEAUTIFUL TCP SERVER BUT NOW WITH A BIGGER FONT

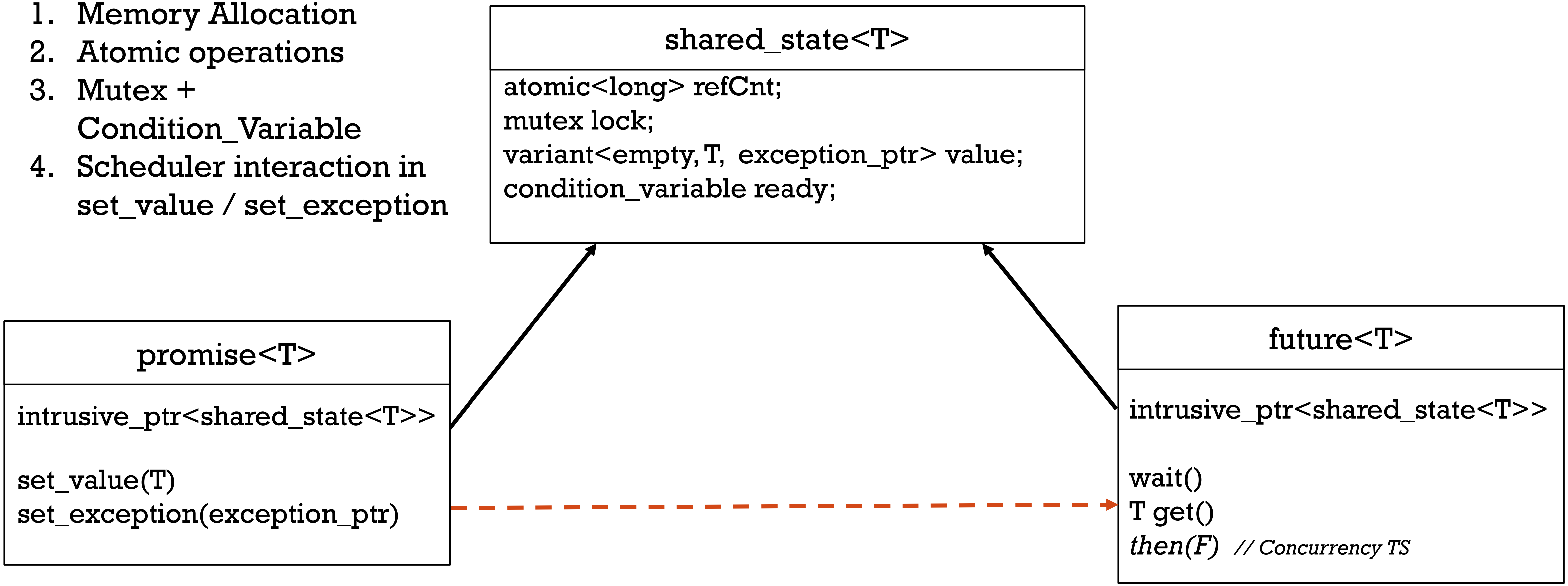
```
std::future<void> session(tcp::socket s, size_t block_size)
{
  s.set_option(tcp::no_delay(true));
  std::vector<char> buf(block_size);

  for(;;) {
    size_t n = co_await async_read_some(s, buffer(buf.data(), block_size));
    n = co_await async_write(s, buffer(buf.data(), n));
  }
}
```

```
std::future<void> server(io_context &io, tcp::endpoint const &endpoint,
                         size_t block_size)
{
  tcp::acceptor acceptor(io, endpoint);
  acceptor.listen();
  for (;;)
    session(co_await async_accept(acceptor), block_size);
}
```

STD::FUTURE<T> AND STD::PROMISE<T>

1. Memory Allocation
2. Atomic operations
3. Mutex + Condition_Variable
4. Scheduler interaction in set_value / set_exception



shared_state<T>

```
atomic<long> refCnt;
mutex lock;
variant<empty, T,  exception_ptr> value;
condition_variable ready;
```

promise<T>

```
intrusive_ptr<shared_state<T>>

set_value(T)
set_exception(exception_ptr)
```

future<T>

```
intrusive_ptr<shared_state<T>>

wait()
T get()
then(F)  // Concurrency TS
```

21

COMPLICATIONS

Cancellation and allocation

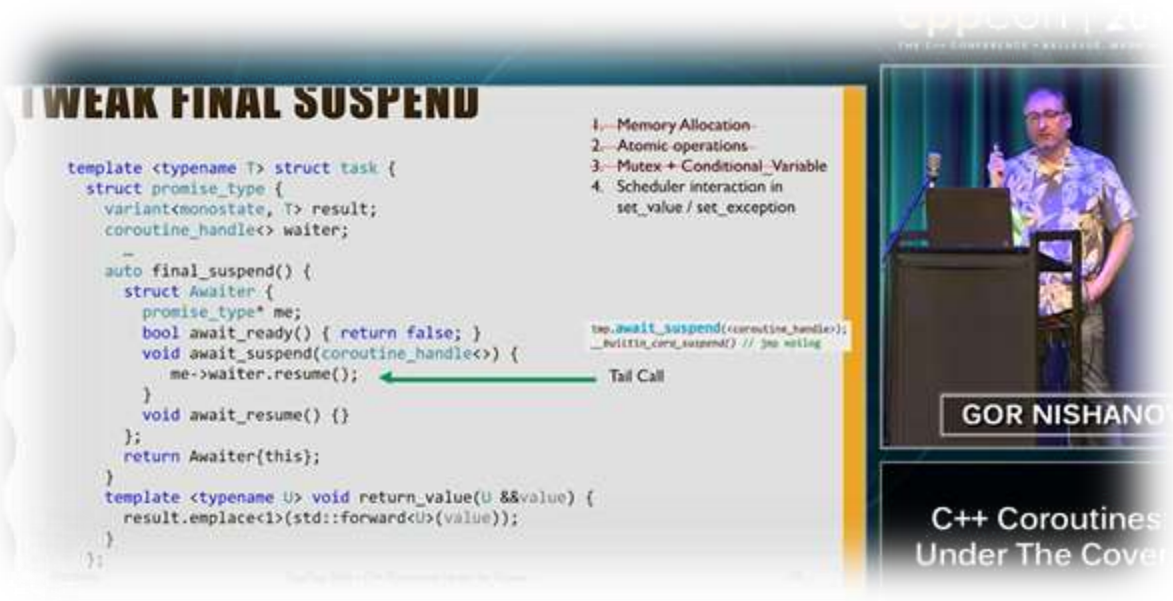
LIVE

23

BEYOND THE TS

Two possible additions to C++ Coroutines

SYMMETRIC CONTROL TRANSFER



WEAK FINAL SUSPEND

```
template <typename T> struct task {
  struct promise_type {
    variant<monostate, T> result;
    coroutine_handle<> waiter;
     …
    auto final_suspend() {
      struct Awaiter {
        promise_type* me;
        bool await_ready() { return false; }
        void await_suspend(coroutine_handle<>) {
           me->waiter.resume();
        }
        void await_resume() {}
      };
      return Awaiter{this};
    }
    template <typename U> void return_value(U &&value) {
      result.emplace<1>(std::forward<U>(value));
    }
  };
```

1. ~~Memory Allocation~~
2. ~~Atomic operations~~
3. ~~Mutex + Conditional_Variable~~
4. Scheduler interaction in set_value / set_exception

```
tmp.await_suspend(<coroutine_handle>);
__builtin_coro_suspend() // jmp waiting
```

Tail Call

- Available only in clang trunk
- Not in MSVC or clang 5
- Not part of the TS (yet)

```
coroutine_handle<> await_suspend(coroutine_handle<>) {
  return me->waiter;
}
```

PEEKING AT COROUTINE ARGUMENTS FROM PROMISE

```
// Coroutine object returned in an usual place
HRESULT f(X x, Y y, Z z, SomeSmartPtr<MyCoro>* p);

// Would like have access to executor in initial_suspend
void g(executor& e);

// Would like to check whether we are cancelled at every
// suspend point
void h(cancellation_token& c);
```

NOT VERY GOOD WORKAROUND

```
struct promise_type {

  template <typename... Args>
  void* operator new(size_t size, Args const&... args) {
    // stash what you need into a thread_local
  }

  promise_type() {
    // get what you wanted out of a thread_local
  }
  ...
};
```

LET PROMISE CONSTRUCTOR PEEK AT ARGS!

```
struct promise_type {

  template <typename... Args>
  promise_type(Args const&... args) {
    // get what you want
  }
  ...
};
```

- Opt-in feature. Empty construct will work fine
- Will observe stable parameters (parameter copies)
- Implicit object parameter passed as a first argument
- Not part of the TS
- Not available in any compiler

CONCLUSION

- Networking and Coroutine TS are great together
- At the moment, for the best performance use “the hard way”
- Hopefully can be addressed before C++20 ships
- Coroutines are available in
  - MSVC 2017 (/await)
  - clang 5.0 (-fcoroutines-ts –stdlib=libc++)
- Networking TS implementation:
  - https://github.com/chriskohlhoff/networking-ts-impl
- Look at good open source coroutine libraries:
  Example: https://github.com/lewissbaker/cppcoro
- Snippets we used during the live part will be available at:
  https://github.com/GorNishanov/await/tree/master/2017_CppCon

QUESTIONS?