ASIO & CORO

Network Programming with C++20 Coroutines

Part I: From sync to async to as-sync

Peter Eisenlohr
peter@eisenlohr.org
github.com/pgit/asio-coro

OVERVIEW

- Introduction
- Part I: From sync to async to as-sync
- Part II: Working with Coroutines
- Epilogue

INTRODUCTION

ASIO

- ASIO (Asynchronous Input/Output) is a cross-platform C++ library for network and low-level I/O programming. It provides a consistent asynchronous model using modern C++.
- P2444 The Asio asynchronous model, Christopher Kohlhoff, 2021 https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2444r0.pdf
- "Powerful but Complex"
- Lowlevel (e.g., no HTTP) – but there are other libraries that build on top of it Boost::{Process, Beast, MySQL, Redis, MQTT5}, asio-grpc, …

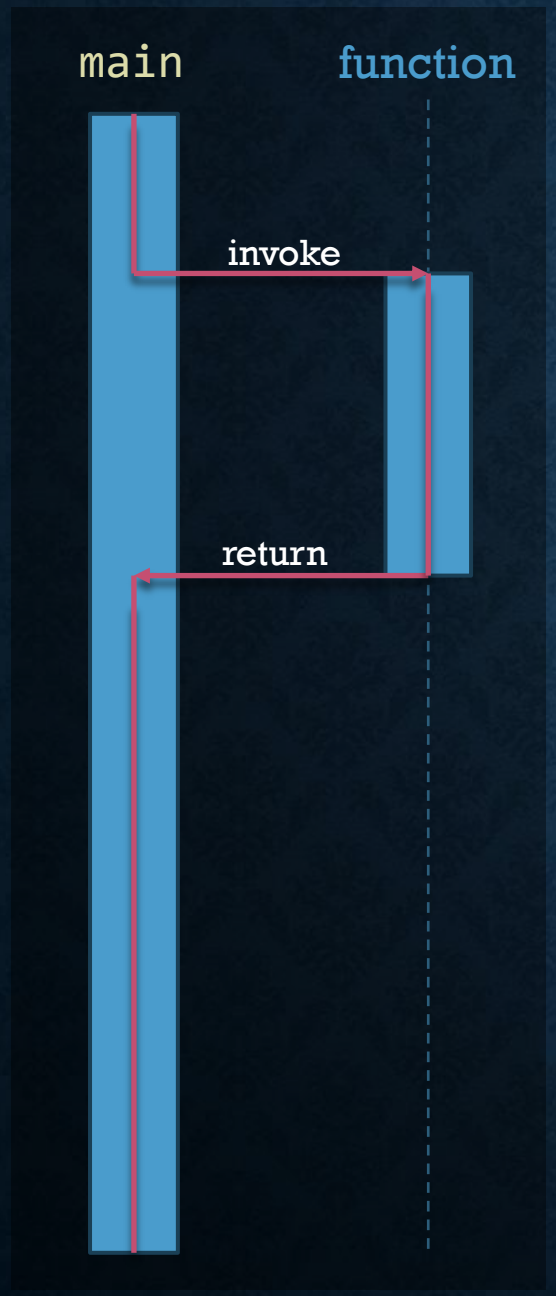
ASIO

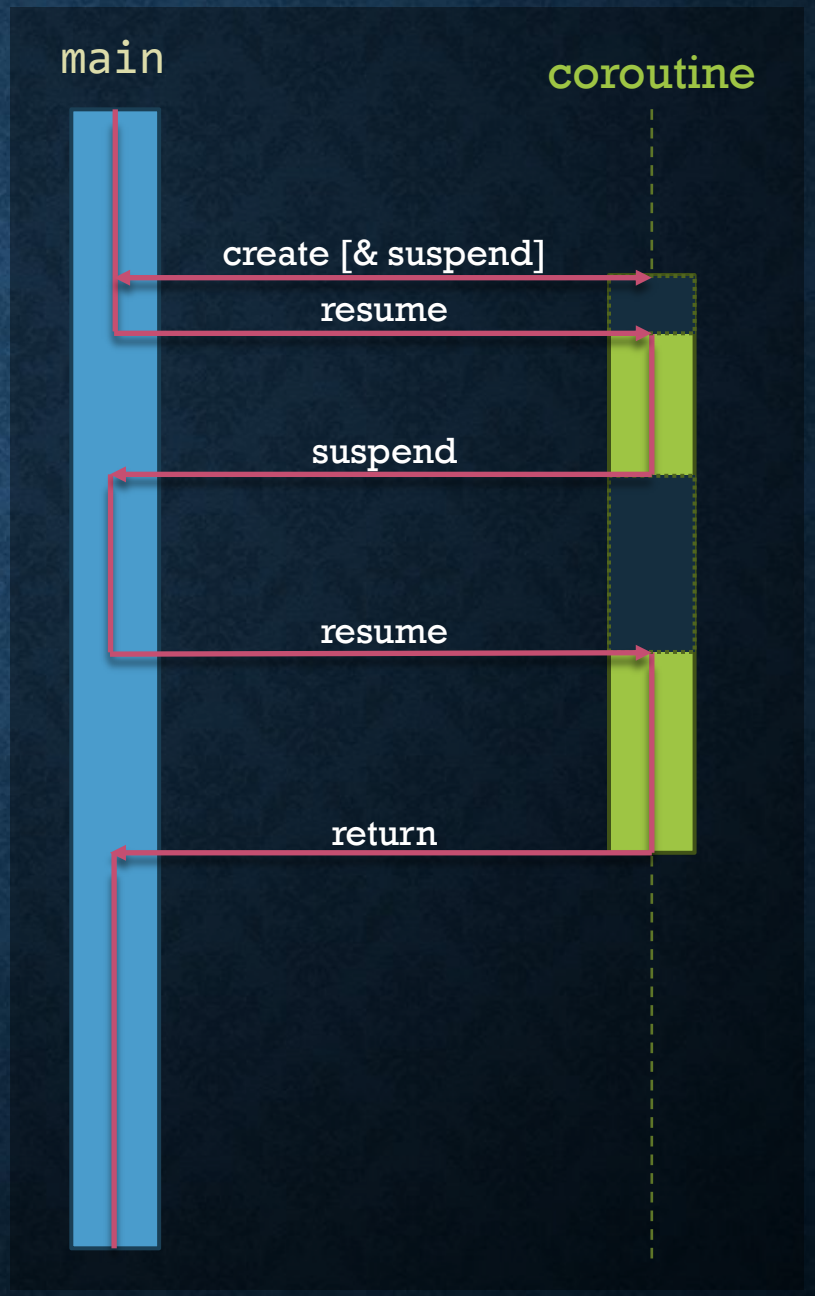
- Key Features:
  - Part of Boost, but also available standalone
  - Supports synchronous and asynchronous operations
  - Works with sockets, timers, serial ports, pipes and more
  - Integrates naturally with C++ coroutines
- Why Use It?
  - Efficient event-driven I/O
  - Scales well for high-performance servers and clients
  - Clean abstraction over platform-specific APIs (epoll, IOCP, etc.)

COROUTINES

- A C++20 coroutine is a special kind of function that can **suspend** and **resume** its execution without blocking the thread.
- New keywords: `co_await`, `co_yield`, and `co_return`.
- Transformed into a state machine by the compiler.
- Enables asynchronous and lazy evaluation patterns.
- Ideal for networking (e.g. with Boost.Asio) where non-blocking IO is crucial.

- DISCLAIMER:
  This presentation is NOT about the C++20 Coroutines Language Feature.


main
function
invoke
return


main
coroutine
create [& suspend]
resume
suspend
resume
return

?

PART I
FROM SYNC TO ASYNC TO AS-SYNC

FROM SYNC TO ASYNC TO AS-SYNC

- Sync Echo Server
- Async Echo Server
- Completion Handlers and Tokens
- The `deferred` completion token
- Coro Echo Server
- `co_spawn`

HELLO WORLD

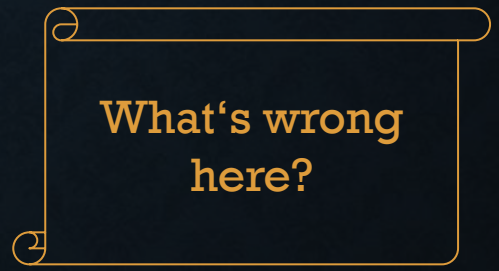
A minimal sync Echo Server

github.com/pgit/asio-coro

```cpp
void session(tcp::socket socket)
{
   std::array<char, 64*1024> data;
   for (;;)
   {
      boost::system::error_code ec;
      size_t n = socket.read_some(buffer(data), ec);
      if (ec == error::eof)
         return;
      write(socket, buffer(data, n));
   }
}
```

```cpp
#include <boost/asio.hpp>

using namespace boost::asio;
using ip::tcp;
```

```cpp
void server(tcp::acceptor acceptor)
{
   for (;;)
      session(acceptor.accept());
}
```




```cpp
int main()
{
   io_context context;
   server(tcp::acceptor{context, {tcp::v6(), 55555}});
}
```

```
void session(tcp::socket socket)
{
   std::array<char, 64*1024> data;
   for (;;)
   {
      boost::system::error_code ec;
      size_t n = socket.read_some(buffer(data), ec);
      if (ec == error::eof)
         return;
      write(socket, buffer(data, n));
   }
}
```

```
void server(tcp::acceptor acceptor)
{
   for (;;)
      std::thread(session, acceptor.accept()).detach();  // was: session(acceptor.accept());
}
```

```
int main()
{
   io_context context;
   server(tcp::acceptor{context, {tcp::v6(), 55555}});
}
```

```
#include <boost/asio.hpp>

using namespace boost::asio;
using ip::tcp;
```

SYNCHRONOUS OPERATIONS PROS & CONS

- Natural control flow (`for`, `if`, `while`, …)
- Can use (blocking) child functions
- Temporary storage is managed naturally (RAII)
- Lifetime of arguments is clear, including the ability to safely pass temporaries
- Allows for Structured Programming within the thread

- Slow – a context switch between threads takes thousands of cycles
- Does not scale well with many connections
- Requires extra synchronziation when accessing shared resources
- Cancellation is difficult

```cpp
void session(tcp::socket socket)
{
   std::array<char, 64*1024> data;
   for (;;)
   {
      boost::system::error_code ec;
      size_t n = socket.read_some(buffer(data), ec);
      if (ec == error::eof)
         return;
      write(socket, buffer(data, n));
   }
}

void server(tcp::acceptor acceptor)
{
   for (;;)
      std::thread(session, acceptor.accept()).detach();
}

int main()
{
   io_context context;
   server(tcp::acceptor{context, {tcp::v6(), 55555}});
}
```

```cpp
#include <boost/asio.hpp>

using namespace boost::asio;
using ip::tcp;
```

?

HELLO ASYNC WORLD

A minimal async Echo Server

github.com/pgit/asio-coro

```
class server;
```

```
#include <boost/asio.hpp>

using boost::system::error_code;
using namespace boost::asio;
using ip::tcp;
```

```
int main()
{
   io_context context;
   server server(tcp::acceptor{context, {tcp::v6(), 55555}});

}
```

```
class server
{
public:
   explicit server(tcp::acceptor acceptor)
      : acceptor_(std::move(acceptor))
      do_accept();
   }

private:
   void do_accept()
   {
      acceptor_.async_accept([this](error_code ec, tcp::socket socket)


         do_accept();
      });
   }

   tcp::acceptor acceptor_;
};
```

Control flow:

- sync
- async

WHAT DID WE JUST DO?

- Unlike `accept` before, `async_accept` returns immediately
- Completion Handler is invoked when operation has completed
- We are no longer in control of the program‘s execution
- Instead, the framework calls us when something happens
- „Hollywood Principle“ – We call you, don‘t call us!
- This is „Inversion of Control“:
- `main()` needs to go into `context.run()`

```cpp
#include <boost/asio.hpp>

using boost::system::error_code;
using namespace boost::asio;
using ip::tcp;
```

```cpp
int main()
{
    io_context context;
    server server(tcp::acceptor{context, {tcp::v6(), 55555}});
    context.run();
}
```

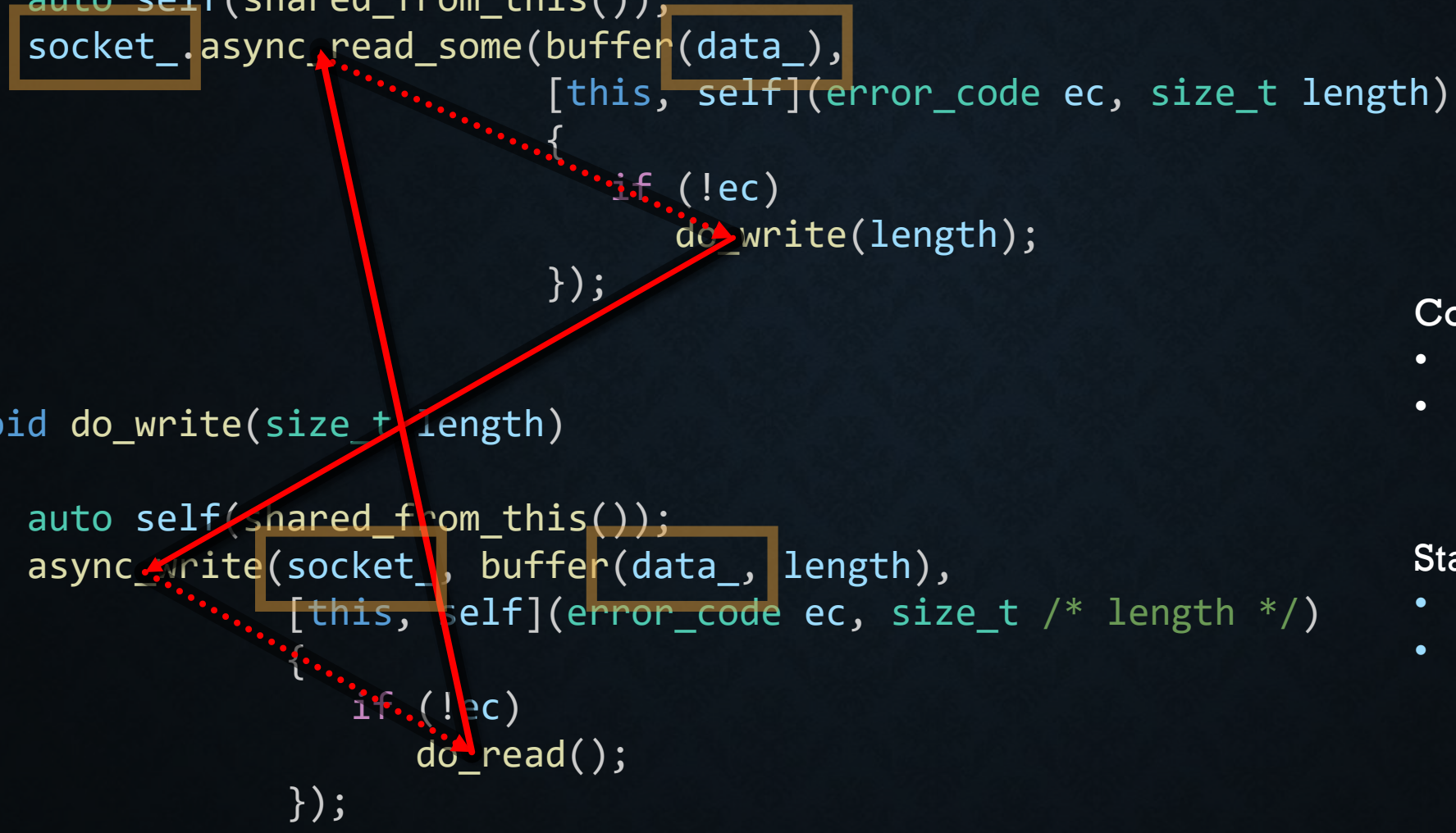
?

```
class server
{
public:
   explicit server(tcp::acceptor acceptor)
      : acceptor_(std::move(acceptor))
      do_accept();
   }

private:
   void do_accept()
   {
      acceptor_.async_accept([this](error_code ec, tcp::socket socket)


         do_accept();
      });
   }

   tcp::acceptor acceptor_;
};
```

```
class server
{
public:
   explicit server(tcp::acceptor acceptor)
      : acceptor_(std::move(acceptor))
      do_accept();
   }

private:
   void do_accept()
   {
      acceptor_.async_accept([this](error_code ec, tcp::socket socket)
         if (!ec)
            std::make_shared<session>(std::move(socket))->start();
         do_accept();
      });
   }

   tcp::acceptor acceptor_;
};
```

```
class session : public std::enable_shared_from_this<session>
{
public:
   session(tcp::socket socket) : socket_(std::move(socket)) {}
   void start() { do_read(); }

private:
   void do_read()
   {
      auto self(shared_from_this());
      socket_.async_read_some(buffer(data_),
                              [this, self](error_code ec, size_t length)
                              {
                                 if (!ec)
                                    do_write(length);
                              });
   }

   void do_write(size_t length) { … }

   tcp::socket socket_;
   std::array<uint8_t, 64*1024> data_;
};
```

```
void do_read()
{
   auto self(shared_from_this());
   socket_.async_read_some(buffer(data_),
                           [this, self](error_code ec, size_t length)
                           {
                              if (!ec)
                                 do_write(length);
                           });
}

void do_write(size_t length)
{
   auto self(shared_from_this());
   async_write(socket_, buffer(data_, length),
               [this, self](error_code ec, size_t /* length */)
               {
                  if (!ec)
                     do_read();
               });
}
```



Control flow:

- sync
- async

State:

- socket_
- data_

HELLO ASYNC WORLD

- Global event loop object: `io_context`
- `run()` function, keeps running the IO context as long as there is work
- Server:
  - initiates `async_accept()`. On completion, spawns a new `session`
- Session:
  - initiates `async_read_some()`, completes with some bytes read
  - initiates `async_write()`, completes after all bytes have been written
  - uses state (`socket_`, `data_`) stored in session class
  - uses `shared_ptr<>` to keep session alive

RECAP: SYNCHRONOUS OPERATIONS PROS & CONS

- Natural control flow (`for`, `if`, `while`, …)
- Can use (blocking) child functions
- Temporary storage is managed naturally (RAII)
- Lifetime of arguments is clear, including the ability to safely pass temporaries
- Allow for Structured Programming

- Slow – a context switch between threads takes thousands of cycles
- Does not scale well with many connections
- Requires extra synchronziation when accessing shared resources
- Cancellation is difficult

ASYNCHRONOUS OPERATIONS PROS & CONS

- Fast – a scheduler designed specifically for events is 1-2 magnitudes faster
- Scales to thousands of connections
- No synchronization required – single thread only
- ❌ Cancellation is still difficult

- ~~Natural control flow (`for`, `if`, `while`, …)~~
- ~~Can use (blocking) child functions~~
- ~~Temporary storage is managed naturally (RAII)~~
- ~~Lifetime of arguments is clear, including the ability to safely pass temporaries~~
- ~~Allow for Structured Programming~~

?

CAN WE DO BETTER?

Yes, we can.

But first let‘s have a closer look at how
Asynchronous Operations
are implemented in ASIO.

COMPLETION HANDLERS AND TOKENS

ASYNCHRONOUS OPERATIONS

IO Object

Initiating Function

Input…

Success / Error

Output…

```
socket.async_read_some(asio::buffer(data), [](error_code ec, std::size_t length) {
   // handler body
});
```

Completion Handler (λ)

Completion Signature

```
void(error_code ec, std::size_t length);
```

ASYNCHRONOUS OPERATIONS

```
timer.async_wait([](error_code ec) {
    // no input, no output
});
```

```
async_write(socket_, buffer(data_, length), [](error_code ec, size_t /* length */) {
    // free function
});
```

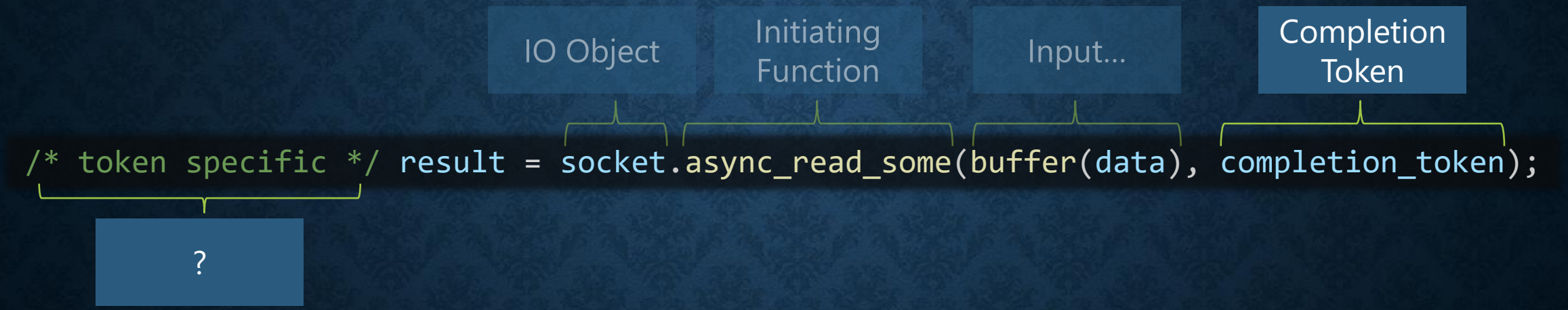
```
post([] {
    // completes unconditionally
});
```

ASYNCHRONOUS OPERATIONS

- „Initiating Function“, doesn‘t run anything yet but schedules for eventual completion
- “Executor” to track work and perform intermediate operations
- „Completion Handler“, invoked when completed (as per post())

- But… the general async model of ASIO can do more
- A „Completion Handler“ is just one type of „Completion Token“

GENERAL INITIATING FUNCTION



COMPLETION TOKENS

Token	Description
Callable (e.g., λ)	The most common token; passes the result directly to the handler.
`use_future`	Returns a `std::future` representing the async result.
`use_awaitable`	Enables C++20 coroutines via `co_await` (returns `awaitable<T>`).
`deferred`	Returns a deferred operation object (lazy async op), which can later be completed by calling `.operator()(handler)` or `operator co_await()`.
`yield_context`	Used for stackful coroutines with Boost.Coroutine (`spawn`).
`detached`	Fire-and-forget: initiates the operation but ignores the result.

COMPLETION TOKEN ADAPTERS

Token Adapter	Description
`as_tuple`	Adapts the token so the completion handler receives results as a single `std::tuple` instead of throwing.
`redirect_error`	Wraps a token and redirects errors to a `boost::system::error_code&` instead of throwing.
`bind_executor`	Binds an executor to a token, ensuring the handler runs on the specified executor.
`bind_allocator`	Binds an allocator to a token for handler memory allocation.
`cancel_{at,after}`	Cancel an operation if not completed before timeout (since v1.86)
`consign`, `append`, `prepend`, …	Rarely used, prefer λ captures

TOKEN: USE_FUTURE

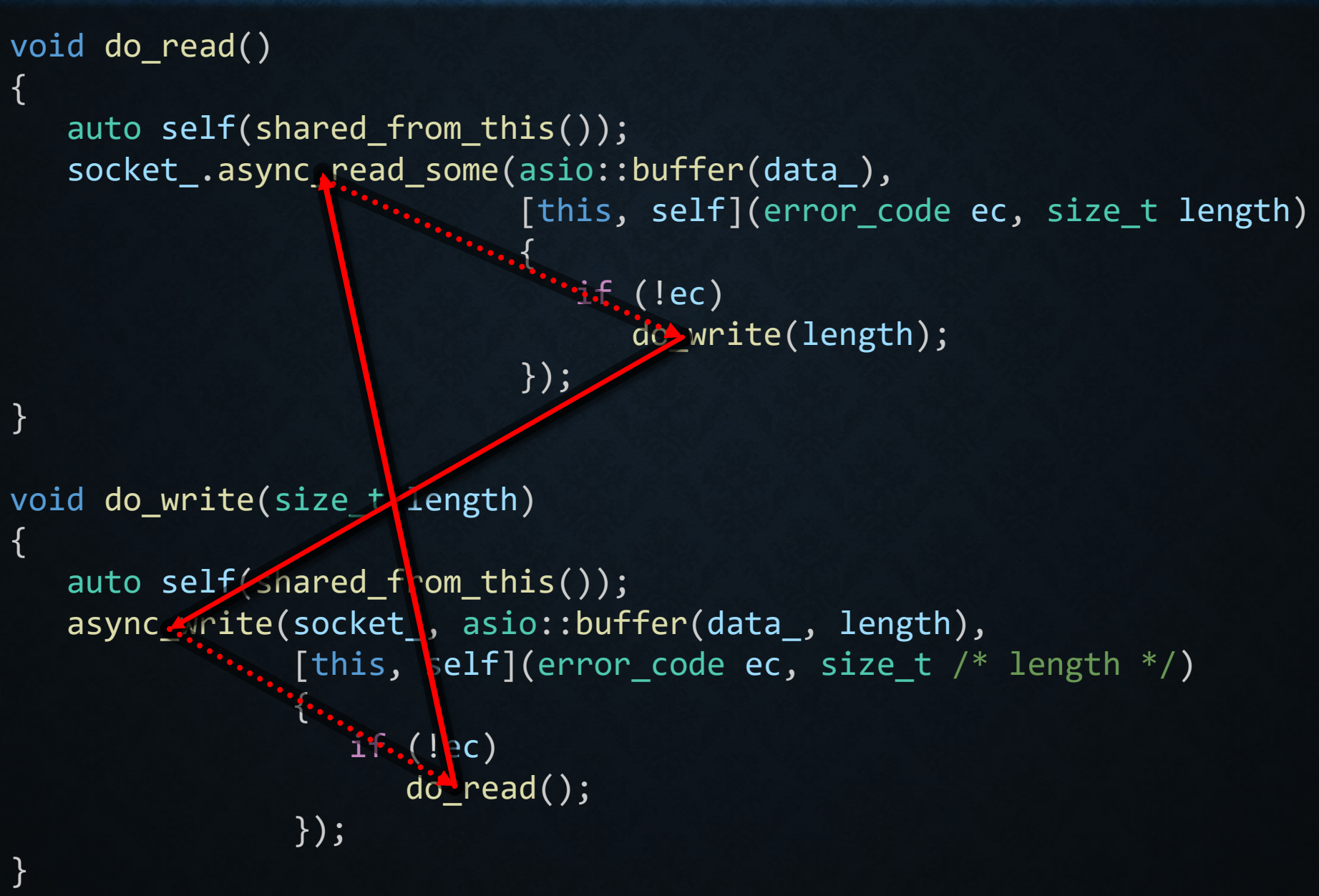
```
auto future = socket.async_read_some(buffer(data), use_future);
size_t n = future.get();  // throws on error
```

TOKEN ADAPTER: AS_TUPLE

```
auto future = socket.async_read_some(buffer(data), as_tuple(use_future));
auto [ec, n] = future.get();
```

?

THE `deferred` COMPLETION TOKEN


void do_read()
{
auto self(shared_from_this());
socket_.async_read_some(asio::buffer(data_),
[this, self](error_code ec, size_t length)
{
if (!ec)
do_write(length);
});
}
void do_write(size_t length)
{
auto self(shared_from_this());
async_write(socket_, asio::buffer(data_, length),
[this, self](error_code ec, size_t /* length */)
{
if (!ec)
do_read();
});
}

```
awaitable<void> session(tcp::socket socket)
{
    std::array<char, 64*1024> data;

    for (;;)
    {
        auto [ec, n] = co_await socket.async_read_some(buffer(data),
                                                       as_tuple(deferred));
        if (ec)
            break;

        std::tie(ec, n) = co_await async_write(socket, buffer(data, n),
                                               as_tuple(deferred));
        if (ec)
            break;
    }
}
```

Let‘s compare…

Coroutines are like Callbacks, in a specific pattern:

- The callback is transformed into the code until the next `co_await`
- Member variables become part of the coroutine frame

```
tcp::socket socket_;
std::array<uint8_t, 64*1024> data_;

void do_read()
{
   auto self(shared_from_this());
   socket_.async_read_some(buffer(data_),
      [this, self](error_code ec, size_t length)
      {
        if (!ec)
           do_write(length);
      });
}

void do_write(size_t length)
{
   auto self(shared_from_this());
   boost::asio::async_write(socket_, buffer(data_, length),
      [this, self](error_code ec, size_t /* length */)
      {
         if (!ec)
            do_read();
      });
}
```

```
awaitable<void> session(tcp::socket socket)
{
   std::array<char, 64*1024> data;

   for (;;)
   {
      auto [ec, n] = co_await socket.async_read_some(buffer(data),
                                                     as_tuple(deferred));

      if (ec)
         break;

      std::tie(ec, n) = co_await async_write(socket, buffer(data, n),
                                             as_tuple(deferred));

      if (ec)
         break;
   }
}
```

```
awaitable<void> session(tcp::socket socket)
{
   std::array<char, 64*1024> data;
   for (;;)
   {
      size_t n = co_await socket.async_read_some(buffer(data), deferred);
      co_await async_write(socket, buffer(data, n), deferred);
   }
}
```

default since v1.86

```
using Socket = deferred_t::as_default_on_t<tcp::socket>;

awaitable<void> session(Socket socket)
{
   std::array<char, 64*1024> data;
   for (;;)
   {
      size_t n = co_await socket.async_read_some(buffer(data));
      co_await async_write(socket, buffer(data, n));
   }
}
```

THE `deferred` COMPLETION TOKEN

- Makes the initiating function return an object that can be `co_await`'ed
- `deferred` is the default completion token as of Boost::Asio Version 1.86
- If the operation completes with an error, that is thrown on `co_await`
- Behavior can be modified using token adapters
- Also supports some other forms of continuations
- Here, the most interesting property is that it is `co_await`'able
- It's lazy – that means that the async operation is not started until awaited

AS-SYNC ASYNCHRONOUS OPERATIONS PROS & PROS

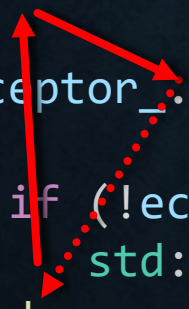
- Fast – a scheduler designed specifically for events is 1-2 magnitudes faster
- Scales to thousands of connections
- No synchronization required – single thread only
- Cancellation works mostly as expected

- Natural control flow (`for`, `if`, `while`, …)
- Can use child ~~functions~~ coroutines
- Temporary storage is managed naturally (RAII)
- Lifetime of arguments is clear, ~~including the ability to safely pass temporaries~~
- Allow for Structured Programming

?

CO_SPAWN

```
class server
{
public:
   explicit server(tcp::acceptor acceptor)
      : acceptor_(std::move(acceptor))
      do_accept();
   }

private:
   void do_accept()
   {
      acceptor_.async_accept([this](error_code ec, tcp::socket socket)
      {
         if (!ec)
            std::make_shared<session>(std::move(socket))->start();
         do_accept();
      });
   }

   tcp::acceptor acceptor_;
};
```



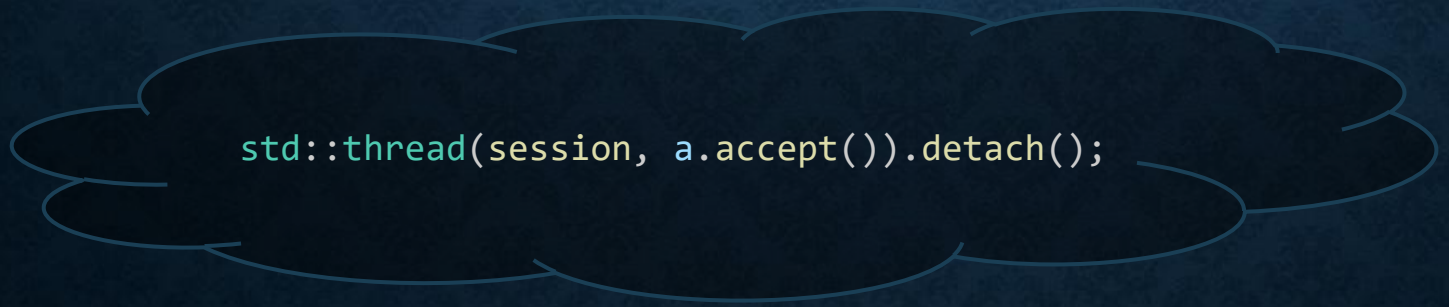
```
awaitable<void> server(tcp::acceptor acceptor)
{
   for (;;)
   {
      auto socket = co_await acceptor.async_accept();
      co_await session(std::move(socket));
   }
}
```

, deferred

What's wrong here?

```
std::thread(session, a.accept()).detach();
```

```
awaitable<void> server(tcp::acceptor acceptor)
{
   for (;;)
   {
      auto socket = co_await acceptor.async_accept();
      co_spawn(acceptor.get_executor(), session(std::move(socket)), detached);
   }
}
```



CO_SPAWN

- ASIO function, not a C++ keyword
- Creates a new coroutine-based "thread of execution"
- `co_spawn` starts an asynchronous operation
- Takes a completion token just as any other initiation function
- Unlike the other async operations, the completion signature is
  - `(std::exception_ptr ep, …)` instead of
  - `(error_code ec, …)`.

```
co_spawn(executor, session(std::move(socket)), detached);
```

CO_SPAWN

Initiating Function

Coroutine

Completion Token

```
/* token specific */ result = co_spawn(executor, awaitable<Result>, completion_token);
```

Output... ?

Completion Signature

```
void(std::exception_ptr ep, Result);
```

```
class server
{
public:
   server(io_context& context, tcp::endpoint endpoint)
      : acceptor_(context, endpoint) {
      do_accept();
   }

private:
   void do_accept()
   {
      acceptor_.async_accept([this](error_code ec, tcp::socket socket)
      {
         if (!ec)
            std::make_shared<session>(std::move(socket))->start();
         do_accept();
      });
   }

   tcp::acceptor acceptor_;
};
```

```
awaitable<void> server(tcp::acceptor a)
{
   for (;;)
      co_spawn(a.get_executor(),
               session(co_await a.async_accept()),
               detached);
}
```

Spawning a coroutine with the `detached` completion token is similar to what we did with the `shared_ptr` and `start()` in the **async** approach.

```
int main()
{
   io_context context;
   co_spawn(context, server({context, {tcp::v6(), 55555}}), detached);
   context.run();
}
```

```cpp
#include <boost/asio.hpp>

using namespace boost::asio;
using ip::tcp;

awaitable<void> session(tcp::socket socket)
{
   std::array<char, 64*1024> data;
   for (;;)
   {
      size_t n = co_await socket.async_read_some(buffer(data));
      co_await async_write(socket, buffer(data, n));
   }
}

awaitable<void> server(tcp::acceptor acceptor)
{
   for (;;)
      co_spawn(acceptor.get_executor(), session(co_await acceptor.async_accept()), detached);
}

int main()
{
   io_context context;
   co_spawn(context, server({context, {tcp::v6(), 55555}}), detached);
   context.run();
}
```

from sync…

```
void session(tcp::socket socket)
{
   std::array<char, 64*1024> data;
   for (;;)
   {
      boost::system::error_code ec;
      size_t n = socket.read_some(buffer(data), ec);
      if (ec == error::eof)
         return;
      write(socket, buffer(data, n));
   }
}

void server(tcp::acceptor a)
{
   for (;;)
      std::thread(session, a.accept()).detach();

}


int main()
{
   io_context context;
   server({context, {tcp::v6(), 55555}});


}
```

… to as-sync

```
awaitable<void> session(tcp::socket socket)
{
   std::array<char, 64*1024> data;
   for (;;)
   {

      size_t n = co_await socket.async_read_some(buffer(data));


      co_await async_write(socket, buffer(data, n));
   }
}

awaitable<void> server(tcp::acceptor a)
{
   for (;;)
      co_spawn(a.get_executor(),
               session(co_await a.async_accept()), detached);
}


int main()
{
   io_context context;
   co_spawn(context, server({context, {tcp::v6(), 55555}}),
                                 detached);
   context.run();
}
```

?

CONCLUSION

- ASIO does event-driven IO, thus „Inversion of Control“
- With callbacks, we lose the ability to reason about our program easily
- By introducing Coroutines, we regain the advantages of Structured Programming

ASIO & CORO

Inversion of Inversion of Control

Peter Eisenlohr
peter@eisenlohr.org
github.com/pgit/asio-coro

RESOURCES

- Boost ASIO Documentation
  - C++20 Examples
  - P2444r0 - The Asio asynchronous model (PDF)
  - https://www.boost.org/doc/libs/1_89_0/doc/html/boost_asio/reference/asynchronous_operations.html
- Blogs
  - Notes on structured concurrency, or: Go statement considered harmful
  - https://cppalliance.org/ruben/2025/04/10/Ruben2025Q1Update.html
  - https://cppalliance.org/asio/2023/01/02/Asio201Timeouts.html
  - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#cpcoro-coroutines
- Repositories
  - https://github.com/pgit/asio-coro (this talk)
  - https://github.com/anarthal/servertech-chat

github.com/pgit/asio-coro/doc/resources.md

RESOURCES

- Using Asio with C++20 Coroutines
  - Asynchrony with ASIO and coroutines - Andrzej Krzemieński - code::dive 2022
  - Cancellations in Asio: a tale of coroutines and timeouts - Rubén Pérez Hidalgo (2025-06)
  - C++20 Coroutines, with Boost ASIO in production: Frightening but awesome
- Chris Kohlhoff
  - Talking Async Ep1: Why C++20 is the Awesomest Language for Network Programming
  - Talking Async Ep2: Cancellation in depth
- Structured Concurrency
  - Writing Safer Concurrent Code with Coroutines... - Lewis Baker - CppCon 2019
- Stack Overflow
  - Whatever user "sehe" says.

github.com/pgit/asio-coro/doc/resources.md