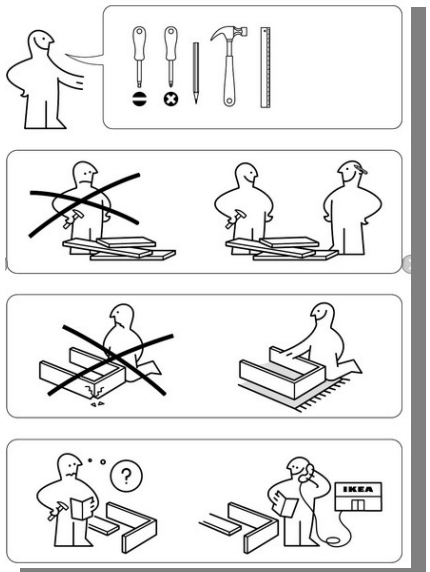
Coroutines

Lieven de Cock
www.codeblocks.org

What we want

What we got



Myths

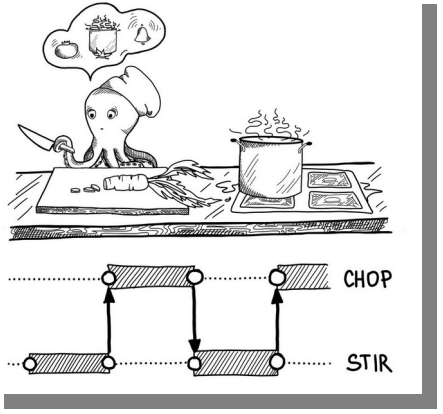
- Is NOT related to:
  - Multi threading
  - Asynchronous programming

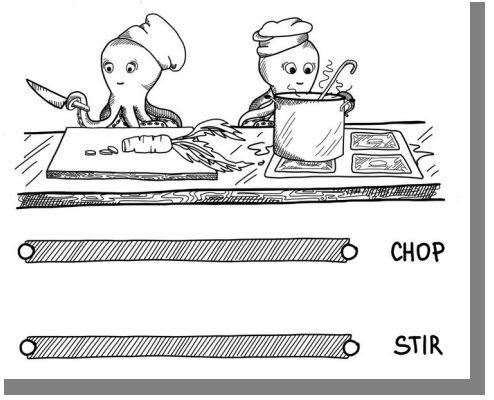
- It can be used in those areas, like it can be used in other areas

Concurrency versus Parallelism

- Processes
- Threads
- Coroutines

- A system is said to be concurrent if it can support two or more actions in progress at the same time. A system is said to be parallel if it can support two or more actions executing simultaneously

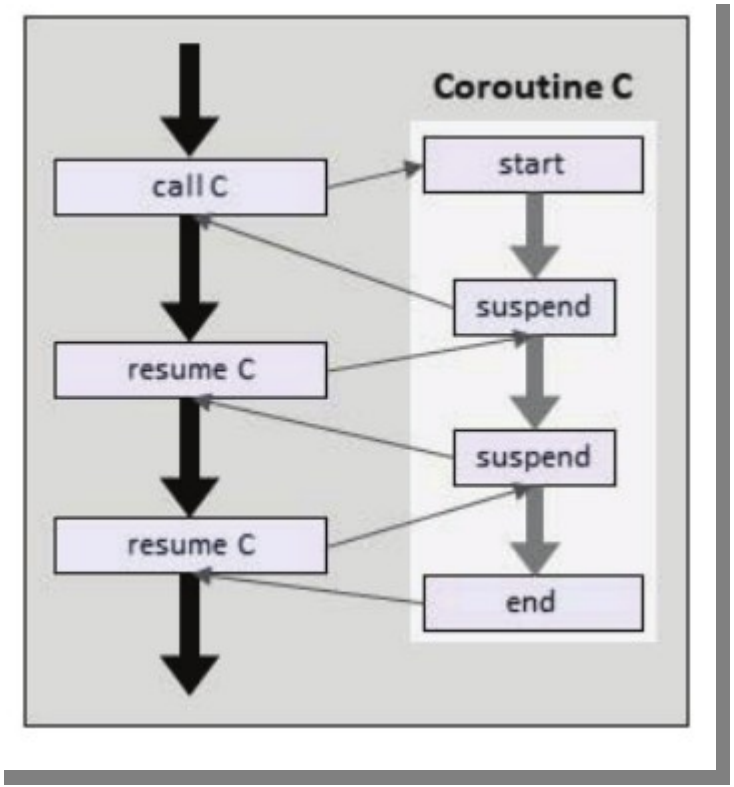




Function / Routine

Function	Coroutine
starts	starts
ends	ends
	suspends
	resumes

Flow



- Returns to the (same) caller
- Same thread (unless ….)

coroutine

- auto foo(…) { … }
  - Is a coroutine when the body contains either:
    - co_return
    - co_yield
    - co_await
  - From that moment the compiler treats it as a coroutine and the magic happens

2 sides to the story

- The compiler facing side
- The user facing side

Radio Station

- At set-up/construction:
  - Type of music
  - How many songs
- A radio station / show emerges, which plays songs
- Modeled by a coroutine
  - Yields a new song and suspends
  - After the user listened to the song, it gets resumed (aka press next)

Song database

```
const int Songs{4};

const std::array<std::string, Songs> electronic
{
    "Front 242 -> No Shuffle",
    "The Neon Judgement -> Tomorrow in the papers",
    "Orbital -> Chime",
    "Underworld -> Born Slippy"
};

const std::array<std::string, Songs> rap
{
    "Public Enemy -> Fight The Power",
    "Ice Cube -> It was a good day",
    "Run DMC -> Walk Thiw Way",
    "Beastie Boys -> Sabotage"
};
```

As a regular function

```
void radioStation(int style, int songs)
{
    std::default_random_engine rng;
    std::uniform_int_distribution<int> track(0, Songs - 1);


    for (int i = 0; i < songs; ++i)
    {
        std::cout << (style ? electronic[track(rng)] : rap[track(rng)]) << std::endl;
    }
}
```

- We get all the songs at the same time, all or nothing
- At the stream out spot we would like to return a song, and later on continue where we left of and provide the next song

The interface towards the user

- Create the coroutine → results in some
  - Coroutine Interface
  - Coroutine api
  - Coroutine remote control
  - …. whatever you want to call it
  - …. something we can further interact with
- Ask for the next song and get it
- Check if there are more songs
- Stop / clean up/ time to go home

This interface sounds more like an object, rather than a function.

So a function (coroutine) needs to result in this “object”.

The function (coroutine has state)

- Let’s assume the music database is a constant global container
- Further (dynamic) state:
  - The type of music
  - The number of songs to play
  - The index in the list of songs (where are we, what’s next, …)
- When suspended, this needs to be stored away
- When resumed, this needs to be restored
- **Stack less** (versus stack full) coroutines
  - The stack frame is stored on the heap by the coroutine mechanism
  - Which as such contains the state described above

Stack frame

- When a function is called some stuff happens
  - Put the return address on the stack
  - Put the function call arguments on the stack
  - Inside the function : local variables are put on the stack
- When the function returns
  - All the above is cleaned up
- The caller continues in its stack frame, and the callee’s stack stuff is gone, as if it was never there

Stack Frame

- We can’t get rid of the stack frame the moment the co-routine yields in the middle
- We need to resume where we left of
- ===> stack frame needs to be restored at that time
- This means:
  - Stack frame needs to be stored somewhere safe (typically on the heap)
  - It needs to be cleaned up at the correct time
    - This is the programmers responsibility !

Stack Frame => coroutine handle

- Stack frame gets put in some allocated memory
- We get a pointer/handle to it
  - Which we need to hold on to
  - Destroy at the appropriate time
    - So at some point during the construction of the co-routine we will get this handle from the eco-system
      - We will come back to this later on

As a couritine

```
RadioStation radioStation(int style, int songs)
{
    std::default_random_engine rng;
    std::uniform_int_distribution<int> track(0, Songs - 1);


    for (int i = 0; i < songs; ++i)
    {
        co_yield( style ? electronic[track(rng)] : rap[track(rng)] );
    }
}
```

- Spot the 2 differences …
  - co_yield
  - Return Value → RadioStation ===> ‘that coroutine object’

RadioStation object : user perspective

- Create it → constructor → the user is NOT directly calling it
- Destroy it → destructor
- Are we done ? → done()
- Next song → next()
  - Obviously not to be called when we are done
  - Coroutine resumes, we get the next song and the coroutine suspends
- Eager/Lazy
  - When created, do we get immediately a song
    - Yes : Eager
    - No : Lazy

We go for lazy, we consider the creation, like plugging the radio into the power socket, we don't turn it on yet (aka play song), so we always explicitly ask for the (first/next) song

RadioStation object : user perspective

- next and done ==> nextSong()
  - Returns true : there is more
  - Returns false : it’s done
- So after resumption we only get signalled, if there was something more
- So need to explicitly fetch our new song

User perspective : looks like

```
int main()
{
    auto radio = radioStation(0, 7);

    while (radio.nextSong())
    {
        std::cout << radio.value() << std::endl;
    }
    return 0;
}
```

Compiler perspective

- Needs to store (computed/return) values
- Needs some configuration:
  - Suspend at startup (eager/lazy)
  - Suspend at the end
  - What to do if an exception occurs
  - Create the “return object”, aka the “api”, aka our RadioStation object
    - When it creates this object it needs to pass in the coroutine handle → constructor argument
- All this is done via the “promise_type”, aka an object (eg. a struct) which adheres to the **promise_type concept**, it lives inside the coroutine handle → coroutine handle is templatized on this promise_type
- promise_type ==> needs to provide *certain* methods, not all possible methods of the concept
- Has nothing to do with std::promise

promise_type : startup : suspend ?

- auto initial_suspend()
  - Yes

```
auto initial_suspend()
{
    return std::suspend_always{};
}
```

  - No

```
auto initial_suspend()
{
    return std::suspend_never{};
}
```

promise_type : at end : suspend ?

- auto final_suspend()
  - Yes

```
auto final_suspend()
{
        return std::suspend_always{};
}
```

  - No

```
auto final_suspend()
{
        return std::suspend_never{};
}
```

promise_type : at exception ?

- void unhandled_exception()

- void unhandled_exception() noexcept

```
{
        std::terminate();
}
```

promise_type : current state of knowledge

```
auto initial_suspend()
{
    return std::suspend_always{};
}

void unhandled_exception()
{
    std::terminate();
}

auto final_suspend() noexcept
{
    return std::suspend_always{};
}
```

promise_type : store value

- When the coroutine **GENERATES** values (co_yield or co_return), it needs to be able to store them somewhere for the user later on to retrieve it ==> in the promise_type
- And when this value is provided → suspend : yes/no ?
- auto yield_value(const T& value)

promise_type : store value

```
auto yield_value(const std::string& valueIn)
{
    value = valueIn;
    return std::suspend_always{};
}
std::string value;
```

promise_type : current state of knowledge

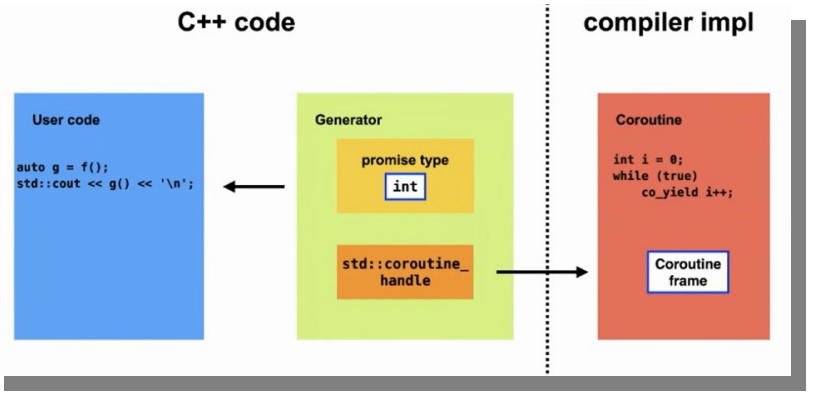
```
auto initial_suspend()
{
    return std::suspend_always{};
}

void unhandled_exception()
{
    std::terminate();
}

auto final_suspend() noexcept
{
    return std::suspend_always{};
}

auto yield_value(const std::string& valueIn)
{
     value = valueIn;
     return std::suspend_always{};
}

std::string value;
```

promise_type : still MISSING

- auto get_return_object()
  - We will come back to this shortly

Hierarchy

- promise_type
  - Is stored in the coroutine HANDLE
- Handle
  - Is stored in the ‘return object/api’ aka RadioStation
- The return object gets the handle at constructor (so it does not create it), and then owns it and (might need) to destroy it



- Image above : (c) Timur Doumler

Handle

- Templatized on the promise_type
  - std::coroutine_handle<promise_type>
- Provides methods to interact with:
  - resume()
  - done()
  - destroy()
  - Is there (still) a handle → if(handle)
  - promise()

RadioStation : return object

```
class [[nodiscard]] RadioStation
{
public:
    struct promise_type;
    using CoroHandle = std::coroutine_handle<promise_type>;

    RadioStation(auto handle) :
        mHandle{handle}
    {
    }

    ~RadioStation()
    {
        if (mHandle)
        {
            mHandle.destroy();
        }
    }

    // …. more ...
private:
    CoroHandle mHandle;
};
```

RadioStation : done or do next

```
bool nextSong() const          /// you could call it: resume()
{
    if (!mHandle || mHandle.done())
    {
        return false;
    }
    mHandle.resume();
    return !mHandle.done();
}
```

RadioStation : done or do next

```
bool nextSong() const
{
    return mHandle ?
        (mHandle.resume(), !mHandle.done())
                : false;
}
```

RadioStation : done or do next

- Bool return value
- True : there is more
- False : nothing left to do, don’t call us anymore, it was nice meeting you

RadioStation : get the value

```
std::string value() const
{
    return mHandle.promise().value;
}
```

promise_type : get_return_object

- auto get_return_object()
  - Called by the compiler to create the return object
  - Needs to accept at constructor time the coroutine handle
  - And whatever other constructor arguments we came up with
- Multiple steps
  1) Create the promise_type
  2) Create the coroutine handle
  3) Create the real return object (Eg. RadioStation)
  4) And return it

promise_type : get_return_object

- Step 1 and Step 2 can be combined
- std::coroutine_handle<promise_type>::from_promise(*this)

```
auto get_return_object()
{
        return RadioStation{CoroHandle::from_promise(*this)};
}
```

Putting it all together : RadioStation class

```
class [[nodiscard]] RadioStation
{
public:
    struct promise_type;
    using CoroHandle = std::coroutine_handle<promise_type>;

    RadioStation(auto handle) :
        mHandle{handle}
    {
    }

    ~RadioStation()
    {
        if (mHandle)
        {
            mHandle.destroy();
        }
    }

    RadioStation(const RadioStation&) = delete;
    RadioStation& operator=(const RadioStation&) = delete;

    bool nextSong() const
    {
        if (!mHandle || mHandle.done())
        {
            return false;  // we are done
        }
        mHandle.resume();
        return !mHandle.done();
    }

    std::string value() const
    {
        return mHandle.promise().value();
    }

private:
    CoroHandle mHandle;
};
```

Putting it all together : promise_type

```
struct RadioStation::promise_type
{
    auto get_return_object()
    {
        return RadioStation{CoroHandle::from_promise(*this)};
    }

    auto initial_suspend()
    {
        return std::suspend_always{};
    }

    void unhandled_exception()
    {
        std::terminate();
    }

    auto yield_value(const std::string& valueIn)
    {
        value = valueIn;
        return std::suspend_always{};
    }

    auto final_suspend() noexcept
    {
        return std::suspend_always{};
    }

    std::string value;
};
```

Putting it all together : use it

```
RadioStation radioStation(int style, int songs)
{
    std::default_random_engine rng;
    std::uniform_int_distribution<int> track(0, Songs - 1);



    for (int i = 0; i < songs; ++i)
    {
    co_yield( style ? electronic[track(rng)] :
rap[track(rng)] );
    }
};
```

```
int main()
{
    auto radio = radioStation(0, 7);
    while (radio.nextSong())
    {
        std::cout << radio.value() <<
std::endl;
    }
    return 0;
}
```

Name dropping

- We have a **GENERATOR**, generating std::string's
- C++23 : std::generator

```
std::generator<char> letters(char first)
{
    for (;; co_yield first++);
}

int main()
{
    for (const char ch : letters('a') | std::views::take(26))
    {
        std::cout << ch << ' ';
    }
    std::cout << '\n';
};
```

A co-routine which does not yield/return anything

- So the co-routine does some work on each resumption
- Might never stop
- **co_await**

- Stupid example: 2 – players – pinball
  - The caller is 1 player (I)
  - The co-routine is the other player (You)
  - We each take turns, we **collaboratively** work on the shared pinball machine
  - We want to win, so we will make the co-routine loose after N turns, aka the co-routine will be done

A co-routine which does not yield/return anything

- The interface/api:
  - Resume method, let’s call it play()
  - No need for a get value method, since it returns/yields nothing
  - We configure N (turns) by specifying an integer to the co-routine invocation

Putting it all together : use it

```
Pinball pinball(int turns)
{
    for (int i = 0; i < turns; ++i)
    {
      std::cout << “Coro’s turn to play.” << std::endl;
      co_await std::suspend_always{};
    }
    std::cout << “Coro looses.” << std::endl;
};
```

```
int main()
{
    auto pball = pinball(4);
    while (pball.play())
    {
        std::cout << “My turn to play.” << std::endl;
    }
    return 0;
}
```

promise_type

- There’s no need to store anything
  - So no member needed
  - No yield_value() needed
- We do need to implement another method in the promise_type: **return_void**
  - Nothing special we want to do here → empty implementation
    void return_void() {}

Putting it all together : promise_type

```
struct Pinball::promise_type
{
    auto get_return_object()
    {
        return Pinball{CoroHandle::from_promise(*this)};
    }
    auto initial_suspend()
    {
        return std::suspend_always{};
    }
    void unhandled_exception()
    {
        std::terminate();
    }
    void return_void()
    {
    }
    auto final_suspend() noexcept
    {
        return std::suspend_always{};
    }
};
```

Putting it all together : Pinball class

```
class [[nodiscard]] Pinball
{
public:
    struct promise_type;
    using CoroHandle = std::coroutine_handle<promise_type>;

    Pinball(auto handle) :
        mHandle{handle}
    {
    }

    ~Pinball()
    {
        if (mHandle)
        {
            mHandle.destroy();
        }
    }

    Pinball(const Pinball&) = delete;
    Pinball& operator=(const Pinball&) = delete;

    bool play() const
    {
        if (!mHandle || mHandle.done())
        {
            return false;  // we are done
        }
        mHandle.resume();
        return !mHandle.done();
    }

private:
    CoroHandle mHandle;
};
```

A co-routine which only returns something at the end, when it’s done

- So the co-routine does some work on each resumption
- And at the very end the outcome is ready and returned
- **co_await , co_return**

- Stupid example: calculating the average of a very large set of values
  - Per resumption the co-routine process a certain amount of the data
  - After several resumptions all the work is done
  - And the result is available and can be returned

A co-routine which only returns something at the end, when it’s done

- The interface/api:
  - Resume method, let’s call it calculate()
  - Again need for a get value method, to retrieve the final outcome, let’s call it getResult()

Putting it all together : use it

```
Average average(const std::vector<int>& numbers)
{
    int sum{};
    for (const auto& number : numbers)
    {
        std::cout << "      Number crunching the next value."
<< std::endl;

        sum += number;
        co_await std::suspend_always{};
    }
    std::cout << "      Finally, all calculated." << std::endl;
    co_return sum / numbers.size();
};
```

```
int main()
{
    const std::vector<int> numbers{100, 200, 100, 200, 100,
200, 100, 200};
    auto aver = average(numbers);

    while (aver.calculate())
    {
        std::cout << "More calculations needed." << std::endl;
    }
    std::cout << "Average : " << aver.getResult() << std::endl;
    return 0;
}
```

promise_type

- Again need to store : the end value
- We do need for that to implement another method in the promise_type to allow to store the value: **return_value**
- promise_type as such again has a **member variable**, to be filled in during that return_value(…) call

Putting it all together : promise_type

```
struct Average::promise_type
{
    auto get_return_object()
    {
        return Average{CoroHandle::from_promise(*this)};
    }

    auto initial_suspend()
    {
        return std::suspend_always{};
    }

    void unhandled_exception()
    {
        std::terminate();
    }

    void return_value(const int& valueIn)
    {
        value = valueIn;
    }

    auto final_suspend() noexcept
    {
        return std::suspend_always{};
    }

    int value;
};
```

Putting it all together : Average class

```
class [[nodiscard]] Average
{
public:
    struct promise_type;
    using CoroHandle = std::coroutine_handle<promise_type>;

    Average(auto handle) :
        mHandle{handle}
    {
    }

    ~Average()
    {
        if (mHandle)
        {
            mHandle.destroy();
        }
    }

    Average(const Average&) = delete;
    Average& operator=(const Average&) = delete;

    bool calculate() const
    {
        if (!mHandle || mHandle.done())
        {
            return false;  // we are done
        }
        mHandle.resume();
        return !mHandle.done();
    }

    int getResult() const
    {
        return mHandle.promise().value();
    }

private:
    CoroHandle mHandle;
};
```

Wait a minute

- Awaitables → the operand of co_await
- Awaiters → specific way to implement an awaitable

- Is another configuration point
- Used whenever co_await and co_yield is called

Awaiter

- 3 methods
  - await_ready()
  - await_suspend(awaitHandle)
  - await_resume()

await_ready

- Called immediately **before** the coroutine is suspended
- Allows as such, for some reason, to decide not to suspend after all
- Returns true → coroutine is NOT suspended
- Typically : return false;
- Use case : suspension depends on some data availability

await_suspend(awaitHandle)

- Called immediately **after** the coroutine is suspended
- Parameter : the handle of the coroutine that was suspended
- What to do next ? Again we could neutralize the suspension if we want. We could even destroy the coroutine …

auto await_resume()

- Called when the coroutine is resumed (after a successful suspension)
- Can return a value
  - The value the co_await expression yields

2 Awaiters we know

- std::suspend_always
  - await_ready returns false
- std::suspend_never
  - await_ready returns true

- The 2 other methods are empty, and await_resume returns void
- Aka both do nothing at suspension and resumption

Application : coroutines calling coroutines

- See later

Application : passing values from suspension back to co-routine

- When the co-routine yields us a value at suspension time, we would like from the caller side, pass in a value ourselves for the next resumption cycle
- Example:
  - When we get the song in our RadioStation
  - We will provide a score of our appreciation of this song when requesting(resuming) the next song

Mechanisms we already know

- Passing values between the 2 sides => promise_type
  - So new member in the promise_type
  - In the API a method for the user to pass in this value
    - The implementation will then store it in the promise_type
- How does the co-routine get's it out of the promise_type ?
  - Return value of co_yield call
  - Custom Awaiter
- Let's inspect these 1 by 1

Api change

- Method needed to pass our value
- Let’s call it : score(…)
- Stores it in the promise_type

```
void RadioStation::score(int scoreIn)
{
    mHandle.promise().score = scoreIn;
}
```

Api change

- Calling it

```
int main()
{
    auto radio = radioStation(0, 7);

    while (radio.nextSong())
    {
        std::cout << radio.value() << std::endl;
        radio.score(10);
    }
    return 0;
}
```

promise_type change

- Member needed
- Different **yield_value** (see later)

```
struct RadioStation::promise_type
{
...

    std::string value;
    int score{};
};
```

Co-routine method

- Get's the value as return value of the co_yield call

```
for (int i = 0; i < songs; ++i)
{
   const int score = co_yield( style ? electronic[track(rng)] : rap[track(rng)] );
   std::cout << "The previous track scored " << score << std::endl;
}
```

promise_type change

- We used to return the Awaiter std::suspend_always
- Now we return an instance of our home made Awaiter (template parameter will be explained shortly)

```
auto yield_value(const std::string& valueIn)
{
    value = valueIn;
    return MyAwaiter<CoroHandle>{};
}
```

Awaiter : remember

- await_resume can return a value
  - The value the co_await expression yields
- So that means in this case our score
  - Which is stored in the promise_type
  - Accessible via the handle
- Let’s assume the handle is a member of our Awaiter, we will see later how it ended up in there

Awaiter

```
template <typename Handle>
struct MyAwaiter
{
    Handle hdl{nullptr};

    bool await_ready() const noexcept
    {
        return false;
    }

/// 1 method missing, see later

    auto await_resume() const noexcept
    {
        return hdl.promise().score;
    }
}
```

Awaiter : remember

- await_suspend receives as argument the handle of the co-routine
  - So it can store it
  - And await_resume can use it

Awaiter

```
template <typename Handle>
struct MyAwaiter
{
    Handle hdl{nullptr};
    bool await_ready() const noexcept
    {
        return false;
    }
    void await_suspend(Handle hdlIn) noexcept
    {
        hdl = hdlIn;
    }
    auto await_resume() const noexcept
    {
        return hdl.promise().score;
    }
};
```

Coro call coro

- Say outer coroutine suspends at certain points
- Somewhere it calls another coroutine
  - We want the inner suspensions to be like it were suspensions of the outer
-

...

coroInner()

...

/// this does not work !

Coro call coro

- We need to explicitly LOOP

```
CoroTask coroOuter()
{
    std::cout << "\t\t coroutine Outer started, calling the inner one" << "\n";

    auto sub = coroInner(3);
    while (sub.resume())
    {
        std::cout << "  Outer(): coroInner() suspended\n";
    }
    std::cout << "\t\t outer done with aling the inner, time to suspend outer\n";
    co_await std::suspend_always{};     // SUSPEND
    std::cout << "\t\t outer done\n";
}
```

Coro call coro : Awaitable

- Have the API/Object of the calling coroutine be an Awaitable
  - This allows : **co_await coroInner(3)**
- With some more boilerplate the suspend/resume of the inner will be to the outside world as if they were from the outer one
- Remember:

```
bool resume() const
{
    if (!mHandle || mHandle.done())
    {
        return false;  // we are done
    }
    mHandle.resume();    /// <=========================================
    return !mHandle.done();
}
```

Coro call coro : Awaitable

- What if ?
  - We would resume the handle from the inner one, from within the resume method of the outer
  - Requires:
    - Outer knowing there is an inner handle
    - Checking if inner done or not, if not, resume that handle, if done, resume own (outer) handle
  - Assume we were able to store that handle in the promise_type (of the Outer) : mSub

Coro call coro : Awaitable

```
bool resume() const
{
    if (!mHandle || mHandle.done())
    {
        return false;
    }

    CoroHandle handle{mHandle};
    while(handle.promise().mSub &&
        ! handle.promise().mSub.done() )
    {
        handle = handle.promise().mSub;
    }

    handle.resume();

    return !mHandle.done();
}
```

Coro call coro : Awaitable

- How to get that sub handle in the promise_type of the Outer ?
- Remember Inner is an Awaitable (handle : mHandle)
  - **void await_suspend(auto awaitHdl)**
- This means that “co_await coroInner(3);” :
  - Will call the Awaiter (Inner coroutine object)
  - Passing in the handle of the coroutine that got suspended (handle of Outer)
    - Inner now knows the handle of Outer
    - Fetch the promise from it
    - Store it’s own handle in it

Coro call coro : Awaitable

```
void await_suspend(auto awaitHdl)
{
    awaitHdl.promise().mSub = mHandle;
}
```

co_await (expression) : more

- Can be an expression resulting in any type (say Foo)
- Requires either:
  - The promise_type has a method: **auto await_transform(Foo x)**
  - Foo has : **auto operator co_await()**
- Both scenarios need to return a real Awaiter

```
co_await 242;

co_await Foo{};
```

await_suspend : return type

- There can be 3 different return types
  - void
  - bool
  - handle

await_suspend : return void

- We can do some extra stuff
- Suspension remains in place

await_suspend : return bool

- We can do some extra stuff
- Suspension remains in place when true is returned, cancelled when false is returned

await_suspend : return coroutine handle

- We can do some extra stuff
- The handle of this (other) coroutine is resumed
- Symmetric transfer
- std::noop_coroutine() is case no other coroutine handle to return
- **CONTINUATION**

Some day : std::future::then

- std::async → std::future
- When the async is done ‘then’ do the next thing:
  - Callable passed to myFuture**.then**(...)
- **CONTINUATION**

Boost asio : completion handler

- When the async operation is finished, the completion handler (token) is executed
- **CONTINUATION**

```
boost::asio::async_read(
    client->mSocket,
    boost::asio::dynamic_buffer(client->mReadBuffer),
    std::bind(completionCondition, std::ref(client->mReadBuffer),
std::placeholders::_1, std::placeholders::_2),
    std::bind(&Server::readHandler, this, client, std::placeholders::_1,
std::placeholders::_2));
```

Continuation

- Idea:
  - CoroOuter calls coroInner
  - When inner is ‘finished’, outer should continue, aka outer is the continuation of inner
    - When inner finished
    - Then continue with outer

Continuation

- How:
  - An Awaiter with await_suspend returning the handle of outer
  - An Awaiter to be invoked from inner, at the ‘appropriate time’
    - When it is finishing
      - ==> auto **final_suspend**()

Teaser : boost::asio and coroutines

```
boost::asio::awaitable<void> echo(
    boost::asio::ip::tcp::socket peer_socket,
    boost::asio::ip::tcp::acceptor acceptor)
{
  std::array<char, 1000> buf;
  for (;;)
  {
    co_await acceptor.async_accept(peer_socket, boost::asio::use_awaitable);
    for (;;)
    {
      const auto [error, len] = co_await peer_socket.async_read_some(
        boost::asio::buffer(buf),
        boost::asio::as_tuple(boost::asio::use_awaitable));
      if (error == boost::asio::error::eof)
      {
        break;
      }
      co_await async_write(
        peer_socket,
        boost::asio::buffer(buf, len),
        boost::asio::use_awaitable);
    }
    peer_socket.close();
  }
}
```

```
boost::asio::io_context ctx;

boost::asio::ip::tcp::socket socket{ctx};
boost::asio::ip::tcp::acceptor acceptor{ctx,

boost::asio::ip::tcp::endpoint{boost::asio::ip::tcp::v4(),
6666}};

boost::asio::co_spawn(
        ctx,
        echo(std::move(socket), std::move(acceptor)),
        boost::asio::detached);

ctx.run();
```

QUESTIONS