Deciphering C++ Coroutines

Mastering Asynchronous Control Flow

ANDREAS WEIS

Deciphering Coroutines - Part 2

Mastering Asynchronous Control Flow

Andreas Weis

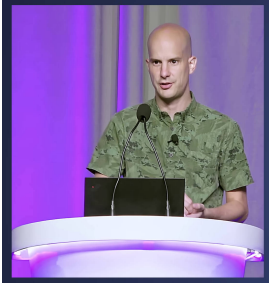
CppCon 2024

About me - Andreas Weis (he/him)

- ComicSansMS
- cpp@andreas-weis.net
- Co-organizer of the Munich C++ User Group

The story so far...

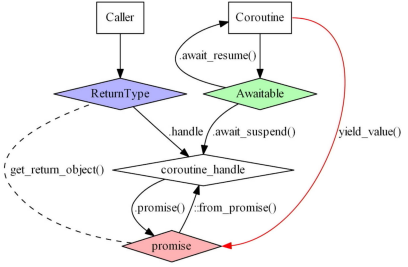
Cppcon 2022 The C++ Conference
September 12th-16th



Andreas Weis

Deciphering Coroutines:
A Visual Approach

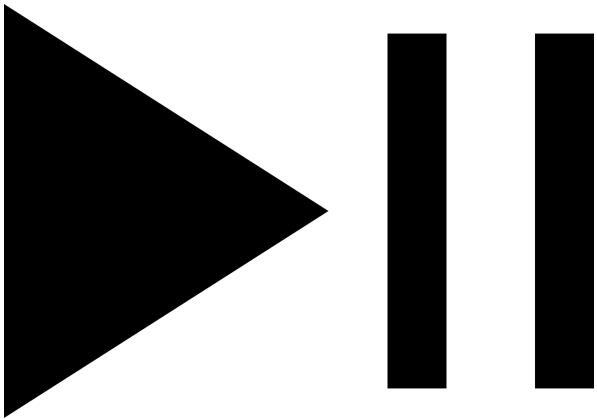
Yielding values



60/68

Video Sponsorship Provided By:
think-cell

Where we left off...



ReturnType

Return type example

```
CustomType my_coroutine();
```

Return type example

```
CustomType my_coroutine();
```

```
std::generator<int> integer_sequence(int begin, int end) {
  for (int i = begin; i < end; ++i) { co_yield i; }
}
```

Return type example

```
CustomType my_coroutine();


std::generator<int> integer_sequence(int begin, int end) {
  for (int i = begin; i < end; ++i) { co_yield i; }
}


int main() {
  for (auto i : integer_sequence(0, 10)) {
    std::println("{}", i);
  }
}
```

The coroutine promise

```
promise_type
```

Cheat Sheet - Promise Type

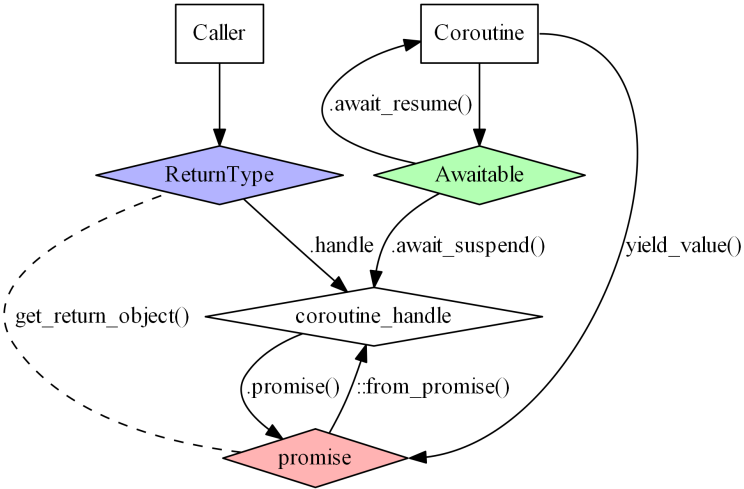
```
1  struct ReturnType / std::coroutine_traits<ReturnType, ...> {
2    struct promise_type {
3      promise_type(T...);   // opt.
4      ReturnType get_return_object();
5      std::suspend_always initial_suspend();
6      // ---- ⇑ Start / ⇓ Shutdown ----
7      void return_value(T); / void return_void();
8      void unhandled_exception();
9      std::suspend_always final_suspend() noexcept;

     };
   };
```

Awaitable

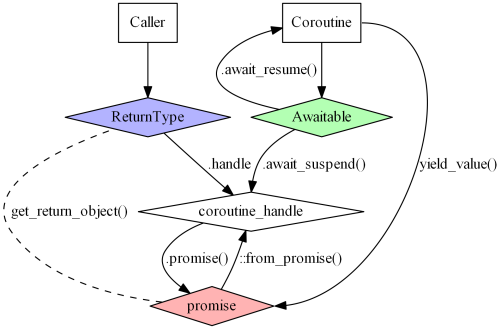
Cheat Sheet - Awaitable

```
1 struct Awaitable {
2   bool await_ready();
3   auto await_suspend(std::coroutine_handle<promise_type>);
4   auto await_resume();
5 };
```

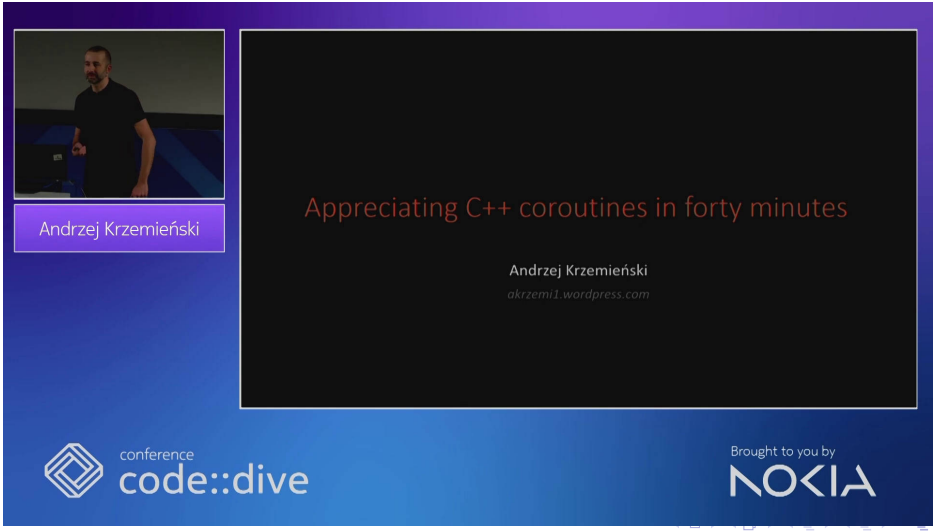
Cheat Sheet: Map of Coroutine Land



Get the cheat sheet!



Andrzej Krzemiesńki at code::dive 2023



Words of caution

- I will at times lead you astray. This is intentional and will hopefully deepen your insight.
- We will be largely ignoring multithreading for this talk.
- This is not a best-practice talk.

A mental model for coroutines: Cooperative Threads

```
void spawn_task() {
  // ...
  Result r = outer_function();
}
```

A mental model for coroutines: Cooperative Threads

```
void spawn_task() {
  // ...
  Result r = outer_function();
}

Result outer_function() {
  PartialResult r = middle_function();
  return Result::from_partial_result(r);
}
```

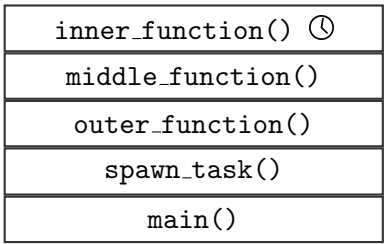
A mental model for coroutines: Cooperative Threads

```
PartialResult middle_function() {
  auto r = inner_function();
  return PartialResult::from_io_result(r);
}
```

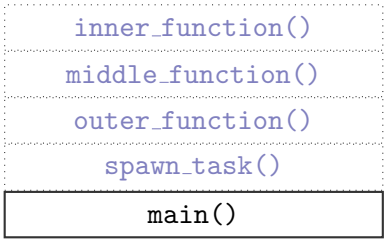
A mental model for coroutines: Cooperative Threads

```
PartialResult middle_function() {
  auto r = inner_function();
  return PartialResult::from_io_result(r);
}

IoResult inner_function() {
  auto data = blocking_io(...); // this could take some time
  return IoResult::from_io_data(data);
}
```

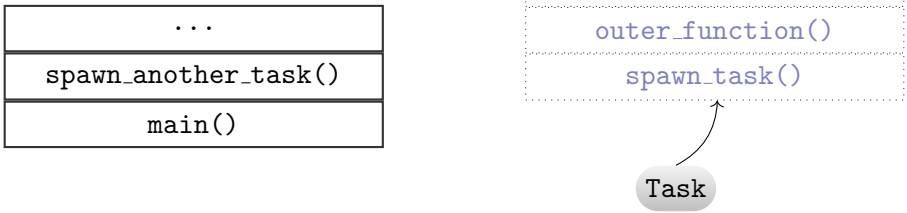
Call Stack


inner_function()
middle_function()
outer_function()
spawn_task()
main()

Call Stack


inner_function()
middle_function()
outer_function()
spawn_task()
main()

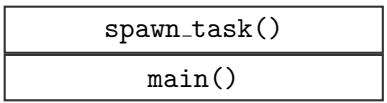
Call Stack


...
spawn_another_task()
main()
inner_function()
middle_function()
outer_function()
spawn_task()
Task

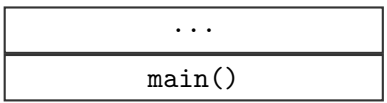
Threads - A straightforward solution

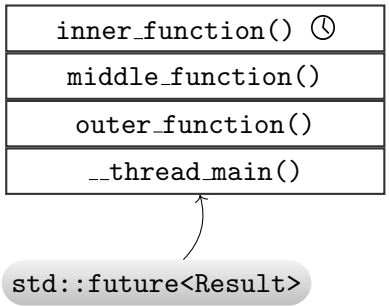
```
std::future<Result> spawn_task() {
  return std::async(outer_function);
}
```

Threads - A straightforward solution



Threads - A straightforward solution





Threads - Evaluation

Threads - Evaluation

- Thread creation is CPU and memory intensive

Threads - Evaluation

- Thread creation is CPU and memory intensive
- Thread switching is expensive

Threads - Evaluation

- Thread creation is CPU and memory intensive
- Thread switching is expensive
- Blocked threads may still consume CPU cycles!

Threads - Evaluation

- Thread creation is CPU and memory intensive
- Thread switching is expensive
- Blocked threads may still consume CPU cycles!
- Shared data must be synchronized correctly.

Threads - Evaluation

- Thread creation is CPU and memory intensive
- Thread switching is expensive
- Blocked threads may still consume CPU cycles!
- Shared data must be synchronized correctly.
- ...

Threads - Evaluation

- Thread creation is CPU and memory intensive
- Thread switching is expensive
- Blocked threads may still consume CPU cycles!
- Shared data must be synchronized correctly.
- ...

Not a solution that scales well.
Can be a good solution for small number of tasks.

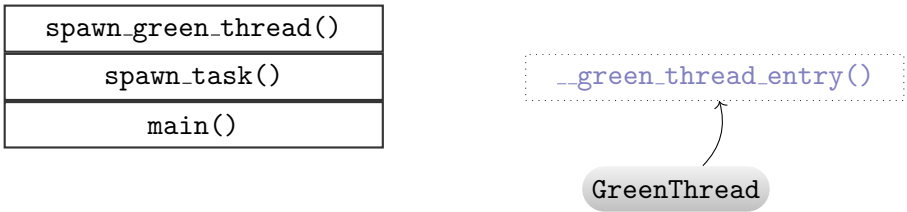
Green Threads aka Stackful Coroutines

```
auto spawn_task() {
  return spawn_green_thread(outer_function);
}
```

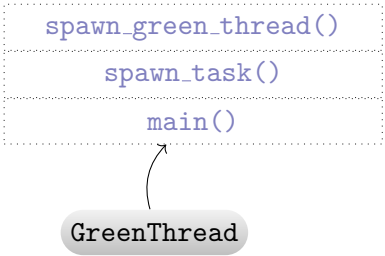
Green Threads aka Stackful Coroutines

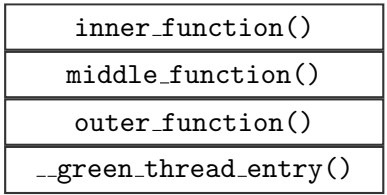
```
auto spawn_task() {
  return spawn_green_thread(outer_function);
}

IoResult inner_function() {
  auto request = setup_non_blocking_io(...);
  this_green_thread::suspend_waiting_for(request);
  auto data = retrieve_io_data(request);
  return IoResult::from_io_data(data);
}
```

Green Threads aka Stackful Coroutines

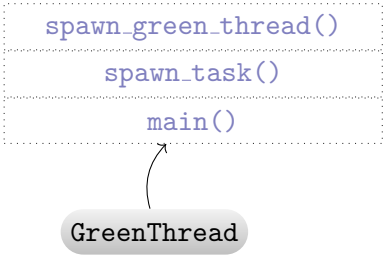


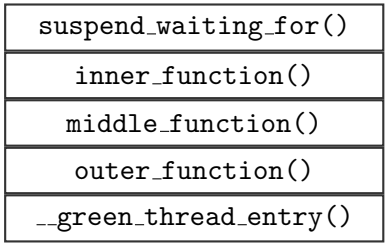
Green Threads aka Stackful Coroutines



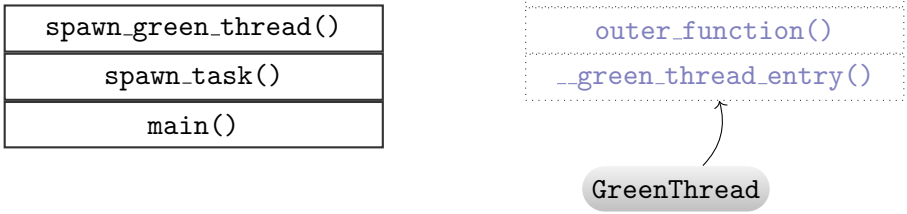


Green Threads aka Stackful Coroutines

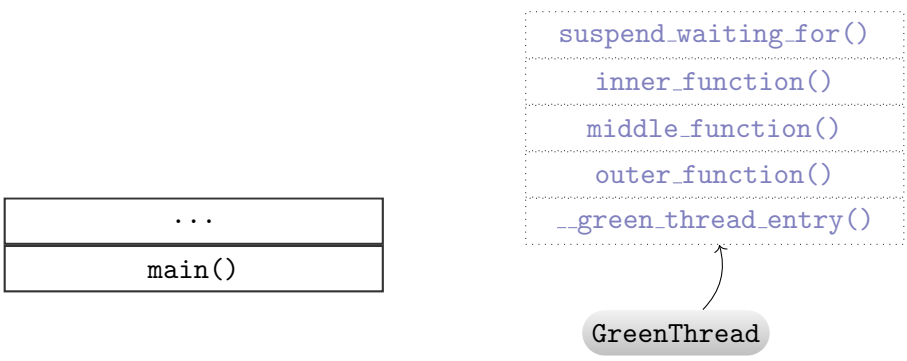




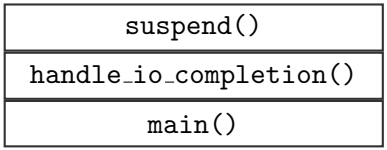
Green Threads aka Stackful Coroutines

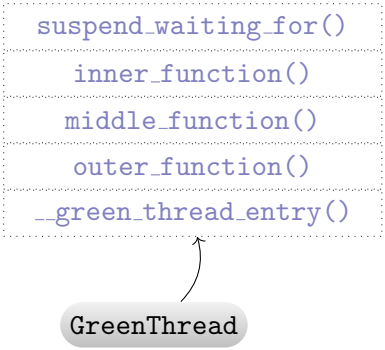


Green Threads aka Stackful Coroutines

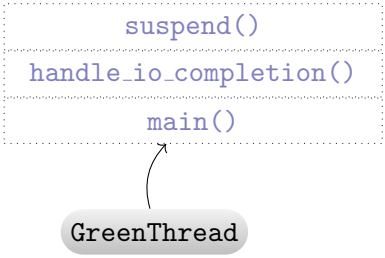


Green Threads aka Stackful Coroutines



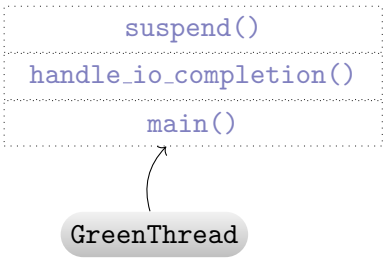


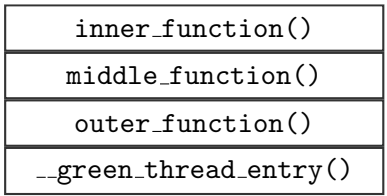
Green Threads aka Stackful Coroutines




```
suspend_waiting_for()
inner_function()
middle_function()
outer_function()
__green_thread_entry()
```

Green Threads aka Stackful Coroutines





Green Threads - Evaluation

- Little overhead

Green Threads - Evaluation

- Little overhead
- No synchronization needed.

Green Threads - Evaluation

- Little overhead
- No synchronization needed.
- Green Threads cooperatively decide when to suspend and resume

Green Threads - Evaluation

- Little overhead
- No synchronization needed.
- Green Threads cooperatively decide when to suspend and resume
- Intuition about control-flow is still very similar to threads

C++20 Coroutines

- Stackless
- We can only suspend one function at a time
- If we want to suspend a computation spanning multiple functions, we need to suspend them all one by one

C++20 Coroutines

- Stackless
- We can only suspend one function at a time
- If we want to suspend a computation spanning multiple functions, we need to suspend them all one by one
- Goal: Suspend execution context larger than a single function

Suspending nested coroutines

```
IoResult inner_function() {
  auto data = blocking_io(...);
  return IoResult::from_io_data(data);
}
```

Suspending nested coroutines

```
IoResult inner_function() {
  auto data = co_await async_io(...);
  co_return IoResult::from_io_data(data);
}
```

Suspending nested coroutines

```
Async<IoResult> inner_function() {
  auto data = co_await async_io(...);
  co_return IoResult::from_io_data(data);
}
```

Suspending nested coroutines

```
Async<IoResult> inner_function() {
  auto data = co_await async_io(...);
  co_return IoResult::from_io_data(data);
}
```

Moving up the stack

```
Async<IoResult> inner_function();


PartialResult middle_function() {
  auto r = inner_function();
  return PartialResult::from_io_result(r);
}
```

Moving up the stack

```
Async<IoResult> inner_function();
```

```
Async<PartialResult> middle_function() {
  auto r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
```

Moving up the stack

```
Async<IoResult> inner_function();
```

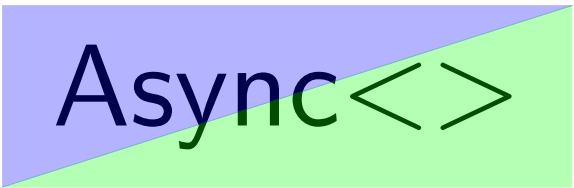
```
Async<PartialResult> middle_function() {
  Async<IoResult> awaitable = inner_function();
  auto r = co_await awaitable;
  co_return PartialResult::from_io_result(r);
}
```

Moving up the stack

```
Async<IoResult> inner_function();
```

```
Async<PartialResult> middle_function() {
  Async<IoResult> awaitable = inner_function();
  auto r = co_await awaitable;
  co_return PartialResult::from_io_result(r);
}
```

Async has two roles to play



Resuming

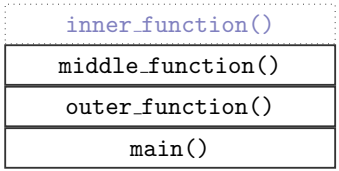
```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();
```

Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();


int main() {
   Async<Result> r = outer_function();
   // ...
   r.resume_computation();
   Result result = r.get_result();
}
```

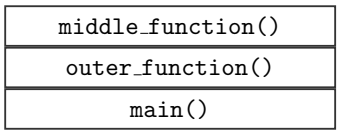
Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();

int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```



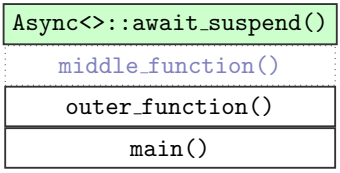
Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();

int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```



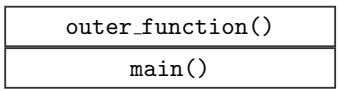
Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();

int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```



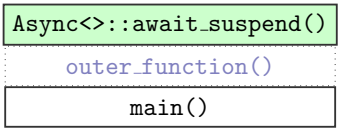
Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();


int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```



Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();

int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```

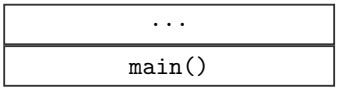


Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();

int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```

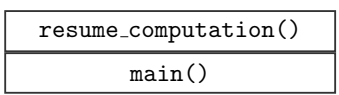
Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();


int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```



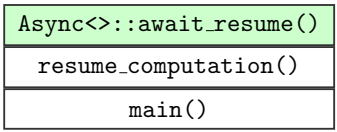
Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();


int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```



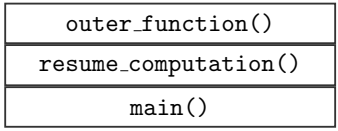
Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();

int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```



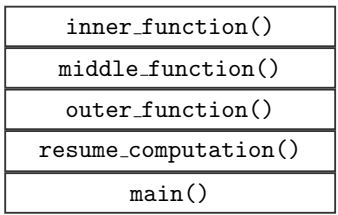
Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();

int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```



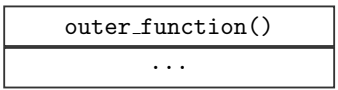
Resuming

```
Async<IoResult> inner_function();
Async<PartialResult> middle_function();
Async<Result> outer_function();


int main() {
  Async<Result> r = outer_function();
  // ...
  r.resume_computation();
  Result result = r.get_result();
}
```



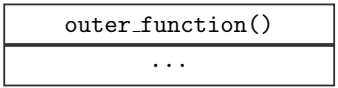
Zooming in...

```
Async<PartialResult> middle_function();

Async<Result> outer_function() {
  Async<PartialResult> awaitable = middle_function();
  PartialResult r = co_await awaitable;
  co_return Result::from_partial_result(r);
}
```



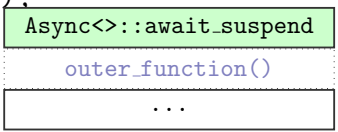
Zooming in...

```
Async<PartialResult> middle_function();

Async<Result> outer_function() {
  Async<PartialResult> awaitable = middle_function();
  PartialResult r = co_await awaitable;
  co_return Result::from_partial_result(r);
}
```



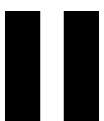
Zooming in...

```
Async<PartialResult> middle_function();

Async<Result> outer_function() {
  Async<PartialResult> awaitable = middle_function();
  PartialResult r = co_await awaitable;
  co_return Result::from_partial_result(r);
}
```



Zooming in...

```
Async<PartialResult> middle_function();

Async<Result> outer_function() {
  Async<PartialResult> awaitable = middle_function();
  PartialResult r = co_await awaitable;
  co_return Result::from_partial_result(r);
}
```

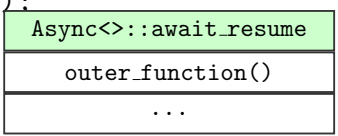


Zooming in...

```
Async<PartialResult> middle_function();

Async<Result> outer_function() {
  Async<PartialResult> awaitable = middle_function();
  PartialResult r = co_await awaitable;
  co_return Result::from_partial_result(r);
}
```

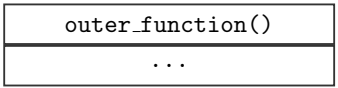
Zooming in...

```
Async<PartialResult> middle_function();

Async<Result> outer_function() {
  Async<PartialResult> awaitable = middle_function();
  PartialResult r = co_await awaitable;
  co_return Result::from_partial_result(r);
}
```



Zooming in...

```
Async<PartialResult> middle_function();

Async<Result> outer_function() {
  Async<PartialResult> awaitable = middle_function();
  PartialResult r = co_await awaitable;
  co_return Result::from_partial_result(r);
}
```



Zooming in...

```
Async<PartialResult> middle_function();

Async<Result> outer_function() {
  Async<PartialResult> awaitable = middle_function();
  PartialResult r = co_await awaitable;
  co_return Result::from_partial_result(r);
}
```

outer_function()
...

Zooming in...

```
Async<PartialResult> middle_function();

Async<Result> outer_function() {
  Async<PartialResult> awaitable = middle_function();
  PartialResult r = co_await awaitable;
  co_return Result::from_partial_result(r);
}
```

Resuming nested suspended coroutines from the outside in does not work!

Resuming nested suspended coroutines from the outside in does not work!

- Once a coroutine has been resumed, it cannot be suspended, unless we call `co_await` again

Resuming nested suspended coroutines from the outside in does not work!

- Once a coroutine has been resumed, it cannot be suspended, unless we call `co_await` again
- We do not know from the outside how often we need to resume an inner function before it succeeds

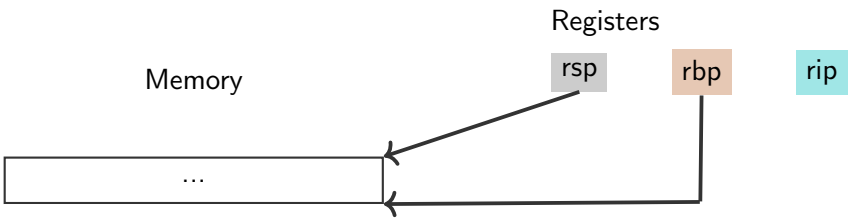
Resuming nested suspended coroutines from the outside in does not work!

- Once a coroutine has been resumed, it cannot be suspended, unless we call `co_await` again
- We do not know from the outside how often we need to resume an inner function before it succeeds
- We want `co_awaiting` an async function to mean: Wake me up once that function has completed and its result is available.

What is a call stack anyway?

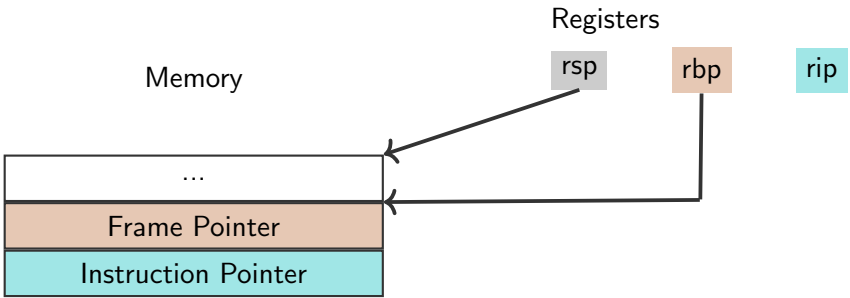
What is a call stack anyway?

Anatomy of a stack frame

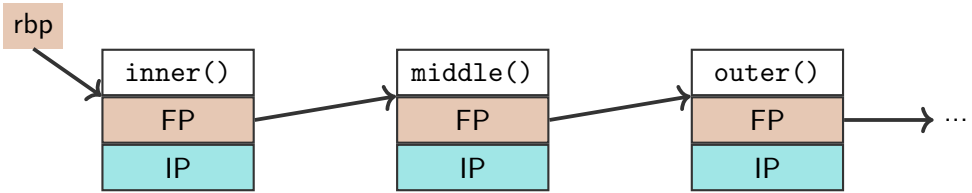


What is a call stack anyway?

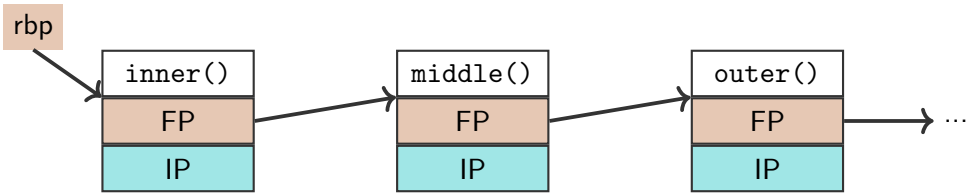
Anatomy of a stack frame



What is a call stack anyway?



What is a call stack anyway?



$\Rightarrow$ Singly-linked list of stack frames with links from callee-frame to caller-frame.

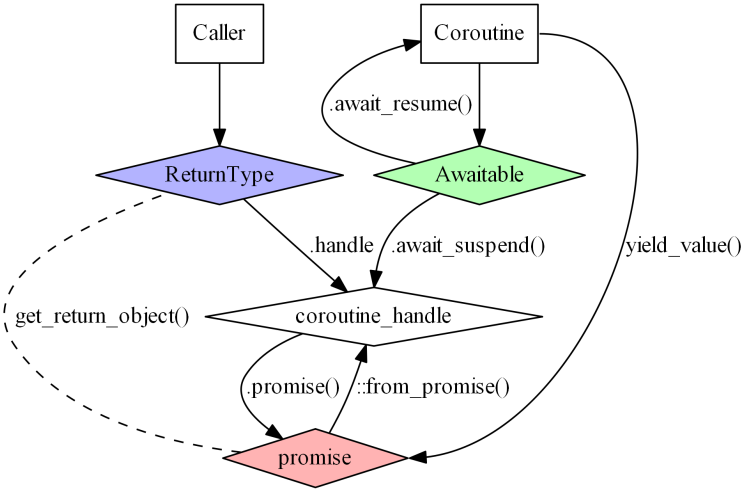
Connecting a coroutine to its caller

```
Async<IoResult> inner_function() {
    // ...
}

Async<PartialResult> middle_function() {
  Async<IoResult> awaitable = inner_function();
  IoResult r = co_await awaitable();
  co_return PartialResult::from_io_result(r);
}
```

Connecting a coroutine to its caller

```
Async<IoResult> inner_function() {
    // ...
}

Async<PartialResult> middle_function() {
  Async<IoResult> awaitable = inner_function();
  IoResult r = co_await awaitable();
  co_return PartialResult::from_io_result(r);
}
```

A look at the cheat sheet...



Setting up the Async object

```
template<typename T> struct Async {
  struct promise_type { /* ... */




  };



};
```

Setting up the Async object

```
template<typename T> struct Async {
  struct promise_type { /* ... */

  };
  std::coroutine_handle<promise_type> self;

};
```

Setting up the Async object

```
template<typename T> struct Async {
  struct promise_type { /* ... */
    Async<T> get_return_object() {
      auto h = std::coroutine_handle<promise_type>
                 ::from_promise(*this);
      return Async<T>{ h };
    }
  };
  std::coroutine_handle<promise_type> self;
  Async<T>(std::coroutine_handle<promise_type> h)
    :self(h)
  {}
};
```

Setting up the Async object

```
template<typename T> struct Async {
  struct promise_type { /* ... */ };

  std::coroutine_handle<promise_type> self;

};
```

Connecting a coroutine to its caller

```
template<typename T> struct Async {
  struct promise_type { /* ... */ };

  std::coroutine_handle<promise_type> self;

};
```

Connecting a coroutine to its caller

```
template<typename T> struct Async {
  struct promise_type { /* ... */ };

  std::coroutine_handle<promise_type> self;

  bool await_ready() { return false; }
  T await_resume() { /* ... */ }
  auto await_suspend(std::coroutine_handle<> handle) {

  }
};
```

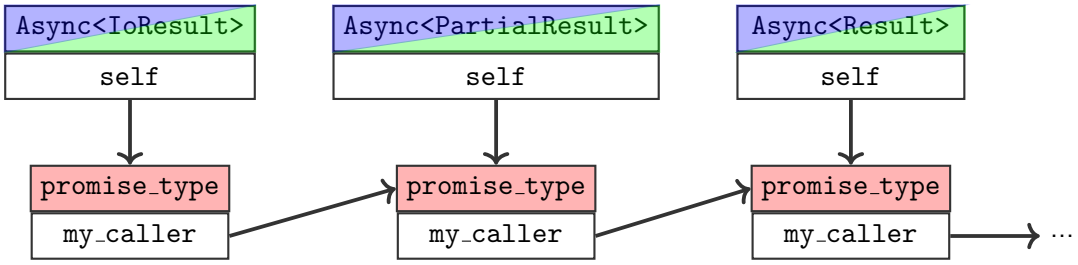
Connecting a coroutine to its caller

```
template<typename T> struct Async {
  struct promise_type { /* ... */ };

  std::coroutine_handle<promise_type> self;

  bool await_ready() { return false; }
  T await_resume() { /* ... */ }
  auto await_suspend(std::coroutine_handle<> handle) {

  }
};
```

Connecting a coroutine to its caller

```
template<typename T> struct Async {
  struct promise_type { /* ... */
    std::coroutine_handle<> my_caller;
  };
  std::coroutine_handle<promise_type> self;


  bool await_ready() { return false; }
  T await_resume() { /* ... */ }
  auto await_suspend(std::coroutine_handle<> handle) {
    self.promise().my_caller = handle;
  }
};
```

We have our own call stack now



Resuming the caller from a nested coroutine

Resuming the caller from a nested coroutine

```
template<typename T>
struct Async {
  struct promise_type { /* ... */
    std::coroutine_handle<> my_caller;


  };
};
```

Resuming the caller from a nested coroutine

```
template<typename T>
struct Async {
  struct promise_type { /* ... */
    std::coroutine_handle<> my_caller;
    auto final_suspend() noexcept {
      return ResumeCaller{};
    }
  };
};

struct ResumeCaller;
```

Resuming the caller from a nested coroutine

```
struct promise_type { /* ... */
  std::coroutine_handle<> my_caller;
};
struct ResumeCaller {




};
```

Resuming the caller from a nested coroutine

```
struct promise_type { /* ... */
  std::coroutine_handle<> my_caller;
};
struct ResumeCaller {
  bool await_ready() { return false; }
  void await_resume() { /* will never be called! */ }




};
```

Resuming the caller from a nested coroutine

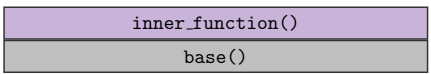
```
struct promise_type { /* ... */
  std::coroutine_handle<> my_caller;
};
struct ResumeCaller {
  bool await_ready() { return false; }
  void await_resume() { /* will never be called! */ }

  coroutine_handle<> await_suspend(
      coroutine_handle<promise_type> h) {

    return h.promise().my_caller;
  }
};
```

Resuming the caller from a nested coroutine

```
struct promise_type { /* ... */
  std::coroutine_handle<> my_caller;
};
struct ResumeCaller {
  bool await_ready() { return false; }
  void await_resume() { /* will never be called! */ }

  coroutine_handle<> await_suspend(
      coroutine_handle<promise_type> h) {
    // Symmetric Transfer!
    return h.promise().my_caller;
  }
};
```

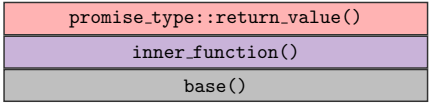
Resuming caller via symmetric transfer

```
Async<> middle_function() {
  auto r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<> inner_function() { /* ... */
  co_return IoResult::from_io_data(data);
}
```



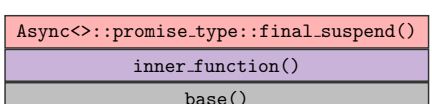
Resuming caller via symmetric transfer

```
Async<> middle_function() {
  auto r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<> inner_function() { /* ... */
  co_return IoResult::from_io_data(data);
}
```



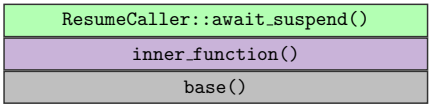
Resuming caller via symmetric transfer

```
Async<> middle_function() {
  auto r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<> inner_function() { /* ... */
  co_return IoResult::from_io_data(data);
}

auto promise_type::final_suspend() {
  return ResumeCaller{};
}
```



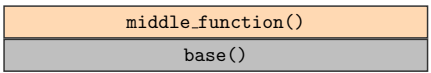
Resuming caller via symmetric transfer

```
Async<> middle_function() {
  auto r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<> inner_function() { /* ... */
  co_return IoResult::from_io_data(data);
}

auto ResumeCaller::await_suspend(
    coroutine_handle<promise_type> h)
  return h.promise().my_caller;
}
```



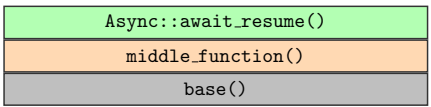
Resuming caller via symmetric transfer

```
Async<> middle_function() {
  auto r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<> inner_function() { /* ... */
  co_return IoResult::from_io_data(data);
}

auto ResumeCaller::await_suspend(
    coroutine_handle<promise_type> h)
  return h.promise().my_caller;
}
```



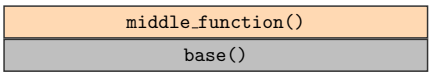
Resuming caller via symmetric transfer

```
Async<> middle_function() {
  auto r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<> inner_function() { /* ... */
  co_return IoResult::from_io_data(data);
}

auto ResumeCaller::await_suspend(
    coroutine_handle<promise_type> h)
  return h.promise().my_caller;
}
```



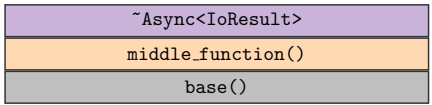
Resuming caller via symmetric transfer

```
Async<> middle_function() {
  auto r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<> inner_function() { /* ... */
  co_return IoResult::from_io_data(data);
}

auto ResumeCaller::await_suspend(
    coroutine_handle<promise_type> h)
  return h.promise().my_caller;
}
```



Resuming caller via symmetric transfer

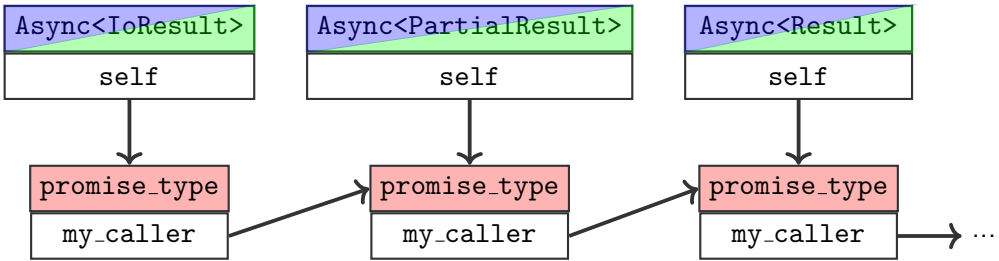
```
Async<> middle_function() {
  auto r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<> inner_function() { /* ... */
  co_return IoResult::from_io_data(data);
}

auto ResumeCaller::await_suspend(
    coroutine_handle<promise_type> h)
  return h.promise().my_caller;
}
```



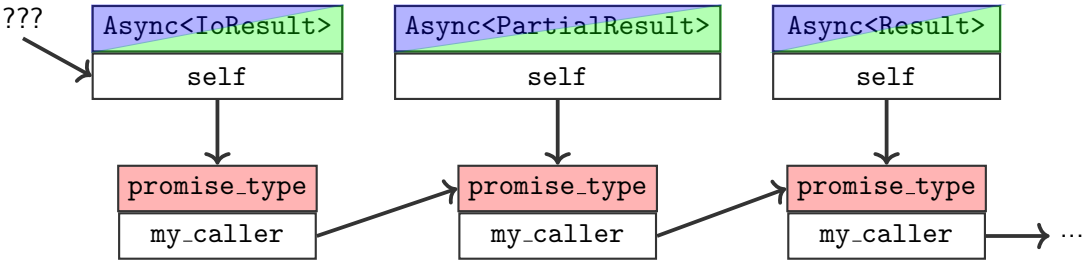
Resuming the caller from a nested coroutine

```
template<typename T> struct Async {
  struct promise_type { /* ... */
    coroutine_handle<> my_caller;
    auto final_suspend() noexcept {
      return ResumeCaller{};
    }
  };
};
struct ResumeCaller { /* ... */
  auto await_suspend(coroutine_handle<promise_type> h) {
    return h.promise().my_caller;
  }
};
```

Resuming the innermost coroutine



Resuming the innermost coroutine



```
Async<IoResult> inner_function() {
  auto data = co_await async_io(...);

  co_return IoResult::from_io_data(data);
}
```

```
Async<IoResult> inner_function() {
  IOAwaitable awaitable = async_io(...);
  auto data = co_await awaitable;
  co_return IoResult::from_io_data(data);
}
```

```
Async<IoResult> inner_function() {
  IOAwaitable awaitable = async_io(scheduler, ...);
  auto data = co_await awaitable;
  co_return IoResult::from_io_data(data);
}
```

Where does the scheduler come from?

```
Async<IoResult> inner_function() {
  auto data = co_await async_io(scheduler, ...);
  /* ... */
}

int main() {
  Scheduler scheduler;
  spawn_task(scheduler);
}
```

Where does the scheduler come from?

```
Async<IoResult> inner_function(Scheduler& scheduler) {
  auto data = co_await async_io(scheduler, ...);
  /* ... */
}

int main() {
  Scheduler scheduler;
  spawn_task(scheduler);
}
```

Passing data from the outside in

```
template<typename T>
struct Async { /* ... */
  struct promise_type { /* ... */
    std::coroutine_handle<> my_caller;

  };

  void await_suspend(std::coroutine_handle<promise_type> h) {
    self.promise().my_caller = h;

  }
}
```

Passing data from the outside in

```
template<typename T>
struct Async { /* ... */
  struct promise_type { /* ... */
    std::coroutine_handle<> my_caller;
    Scheduler* my_scheduler;
  };

  void await_suspend(std::coroutine_handle<promise_type> h) {
    self.promise().my_caller = h;
    self.promise().my_scheduler = h.promise().my_scheduler;
  }
}
```

Passing data from the outside in

```
template<typename T>
struct Async { /* ... */
  struct promise_type { /* ... */
    std::coroutine_handle<> my_caller;
    Scheduler* my_scheduler;
  };

  void await_suspend(std::coroutine_handle<promise_type> h) {
    self.promise().my_caller = h;
    self.promise().my_scheduler = h.promise().my_scheduler;
  }
}
```

Handling different promise types

```
template<typename OtherPromise_T>
void await_suspend(std::coroutine_handle<OtherPromise_T> h) {
  self.promise().my_caller = h;
  self.promise().my_scheduler = h.promise().my_scheduler;
}
```

Handling different promise types

```
template<MyPromiseConcept OtherPromise_T>
void await_suspend(std::coroutine_handle<OtherPromise_T> h) {
  self.promise().my_caller = h;
  self.promise().my_scheduler = h.promise().my_scheduler;
}
```

Spawning up a stack of nested lazy tasks

```
template<typename T> struct Async { /* ... */
  struct promise_type { /* ... */
    std::suspend_always initial_suspend() { return {}; }
  };
};
Async<PartialResult> middle_function();



Async<Result> outer_function();
```

.

Spawning up a stack of nested lazy tasks

```
template<typename T> struct Async { /* ... */
  struct promise_type { /* ... */
    std::suspend_always initial_suspend() { return {}; }
  };
};
Async<PartialResult> middle_function() {
  IoResult r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<Result> outer_function() {
  PartialResult r = co_await middle_function();
  co_return Result::from_partial_result(r);
}
.
```

Spawning up a stack of nested lazy tasks

```
template<typename T> struct Async { /* ... */
  struct promise_type { /* ... */
    std::suspend_always initial_suspend() { return {}; }
  };
};
Async<PartialResult> middle_function() {
  IoResult r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<Result> outer_function() {
  PartialResult r = co_await middle_function();
  co_return Result::from_partial_result(r);
}
.
```

Spawning up a stack of nested lazy tasks

```
template<typename T> struct Async { /* ... */
  struct promise_type { /* ... */
    std::suspend_always initial_suspend() { return {}; }
  };
};
Async<PartialResult> middle_function() {
  IoResult r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<Result> outer_function() {
  PartialResult r = co_await middle_function();
  co_return Result::from_partial_result(r);
}
.
```

Spawning up a stack of nested lazy tasks

```
template<typename T> struct Async { /* ... */
  struct promise_type { /* ... */
    std::suspend_always initial_suspend() { return {}; }
  };
};
Async<PartialResult> middle_function() {
??? IoResult r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<Result> outer_function() {
  PartialResult r = co_await middle_function();
  co_return Result::from_partial_result(r);
}
.
```

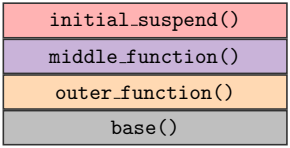
Symmetric transfer to the rescue

```
template<MyPromiseConcept OtherPromise_T>
void await_suspend(std::coroutine_handle<OtherPromise_T> h) {
  self.promise().my_caller = h;
  self.promise().my_scheduler = h.promise().my_scheduler;

}
```

Symmetric transfer to the rescue

```
template<MyPromiseConcept OtherPromise_T>
auto await_suspend(std::coroutine_handle<OtherPromise_T> h) {
  self.promise().my_caller = h;
  self.promise().my_scheduler = h.promise().my_scheduler;
  return self;
}
```

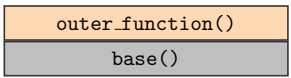
Symmetric transfer to the rescue



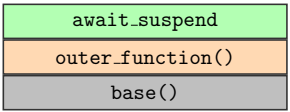

```
Async<PartialResult> middle_function() {
  IoResult r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<Result> outer_function() {
  PartialResult r = co_await middle_function();
  co_return Result::from_partial_result(r);
}
```

Symmetric transfer to the rescue

```
Async<PartialResult> middle_function() {
  IoResult r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<Result> outer_function() {
  PartialResult r = co_await middle_function();
  co_return Result::from_partial_result(r);
}
```



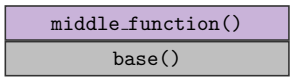
Symmetric transfer to the rescue



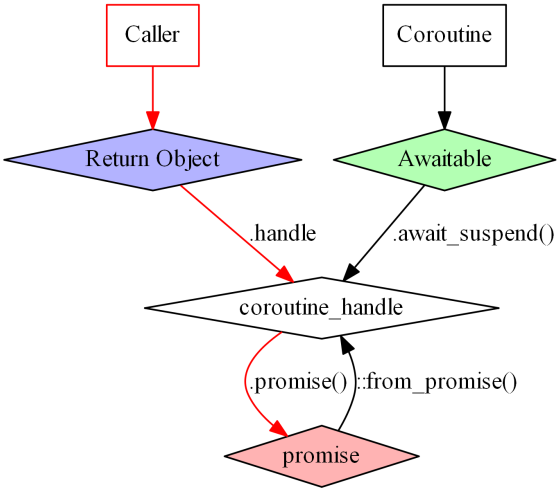

```
Async<PartialResult> middle_function() {
  IoResult r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<Result> outer_function() {
  PartialResult r = co_await middle_function();
  co_return Result::from_partial_result(r);
}
```

Symmetric transfer to the rescue

```
Async<PartialResult> middle_function() {
  IoResult r = co_await inner_function();
  co_return PartialResult::from_io_result(r);
}
Async<Result> outer_function() {
  PartialResult r = co_await middle_function();
  co_return Result::from_partial_result(r);
}
```



Passing a scheduler along



Passing a scheduler along

```
struct promise_type { /* ... */
  Scheduler* scheduler;
};
```

.

Passing a scheduler along

```
struct promise_type { /* ... */
  Scheduler* scheduler;
};

template<MyPromiseConcept OtherPromise_T>
void await_suspend(std::coroutine_handle<OtherPromise_T> h) {
  self.promise().my_caller = h;
  h.promise().my_child = self;

  self.promise().scheduler = h.promise().scheduler;
}
```

But why stop at the scheduler?

```
struct promise_base {
  promise_base* my_caller;
  promise_base* my_child;
};

template<typename T> struct Async { /* ... */
  struct promise_type : promise_base { /* ... */ };
};
```

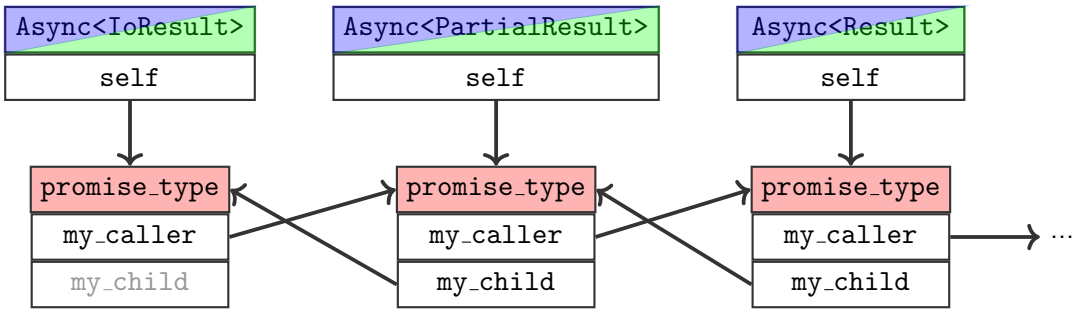
But why stop at the scheduler?

```
template<typename T>
concept MyPromise = std::derived_from<T, promise_base>;

template<typename T> struct Async { /* ... */
  template<MyPromise OtherPromise_T>
  auto await_suspend(std::coroutine_handle<OtherPromise_T> h)
    // establish a doubly-linked list
    self.promise().my_caller = &(h.promise());
    h.promise().my_child = &(self.promise());
    return self;
  }
};
```

But why stop at the scheduler?

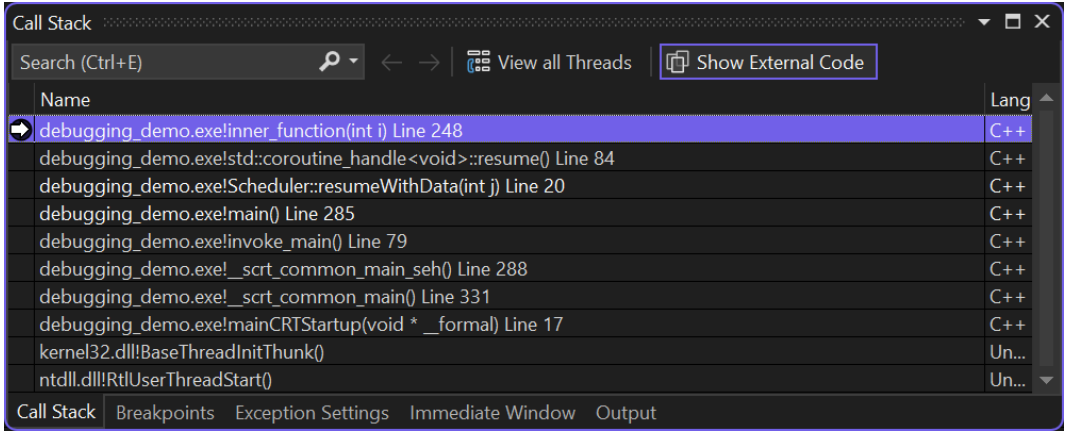
```
struct promise_base {
  promise_base* my_caller;
  promise_base* my_child;

  AnythingYouWant data;
};
```

The call stack is now just another data structure



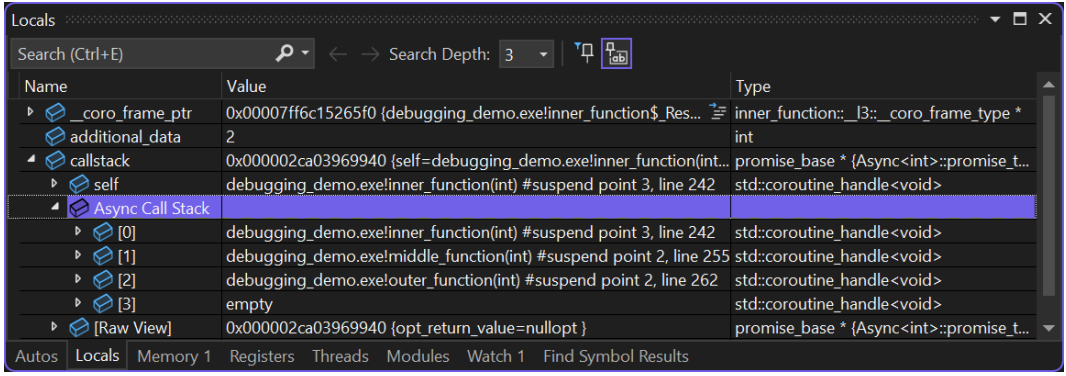
Demo: Injecting scheduler and debugger inspection

https://godbolt.org/z/d7EPTGTdd

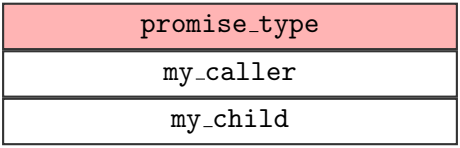


Demo: Injecting scheduler and debugger inspection

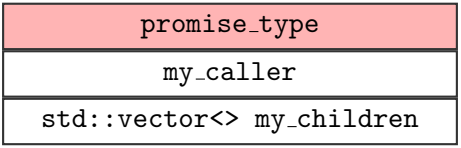
https://godbolt.org/z/d7EPTGTdd



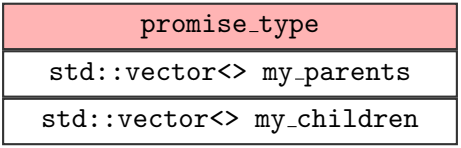
The call stack is now just another data structure



The call stack is now just another data structure



The call stack is now just another data structure



Conclusion

- Coroutines allow for extremely powerful manipulation of control flow between functions
- To achieve best results, there should be some amount of uniformity between coroutine types
- C++26 senders/receivers are a first step in this direction, but a lot is still in development there
- C++ Coroutines are much more flexible than async/await in other languages. It is not yet clear what the best practices are for working with such a flexible feature.
- The community needs more people looking at this, trying interesting things, and joining the conversation

References

Thanks to Lewis Baker and Mateusz Pusz for valuable feedback and discussions on this talk. Any remaining mistakes are my own.

Other related talks at CppCon 2024:

- Dietmar Kühl - Creating a Sender/Receiver HTTP Server
- Ian Petersen, Jessica Wong - How Meta Made Debugging Async Code Easier with Coroutines and Senders
- Dmitry Prokoptsev - Coroutines and Structured Concurrency in Practice
- Michael Caisse - Sender Patterns to Wrangle Concurrency in Embedded Devices

Thanks for your attention.

ComicSansMS / cpp@andreas-weis.net