Coroutines

All you need to know about coroutines

Dawid Pilarski

dawid.pilarski@panicsoftware.com
blog.panicsoftware.com

Agenda

Coroutine theory - what are coroutines?

Practical part I - using the cppcoro library

Theory - promise_types

Promise_type exercise

Asynchronous coroutines

Asynchronous coroutines - exercises

Coroutine theory - what are coroutines?

What are coroutines?

Coroutines are generalization of the function, that can be:

- created

What are coroutines?

Coroutines are generalization of the function, that can be:

- created
- called

What are coroutines?

Coroutines are generalization of the function, that can be:

- created
- called
- returned from

What are coroutines?

Coroutines are generalization of the function, that can be:

- created
- called
- returned from
- suspended

What are coroutines?

Coroutines are generalization of the function, that can be:

- created
- called
- returned from
- suspended
- resumed

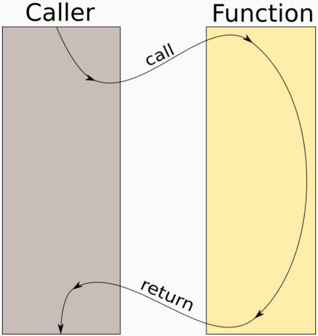
What are coroutines?

Coroutines are generalization of the function, that can be:

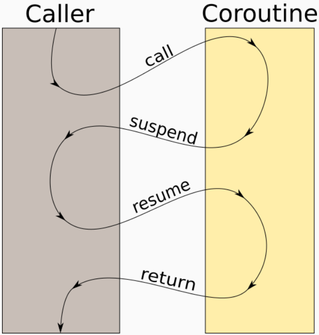
- created
- called
- returned from
- suspended
- resumed
- destroyed

Coroutine flowchart

Function's flow:



Coroutine flow:



What are coroutines for?

Common use cases for coroutines are:

- lazy computation of the sequences (generators)

What are coroutines for?

Common use cases for coroutines are:

- lazy computation of the sequences (generators)
- possibility of introducing asynchronous code with one thread

What are coroutines for?

Common use cases for coroutines are:

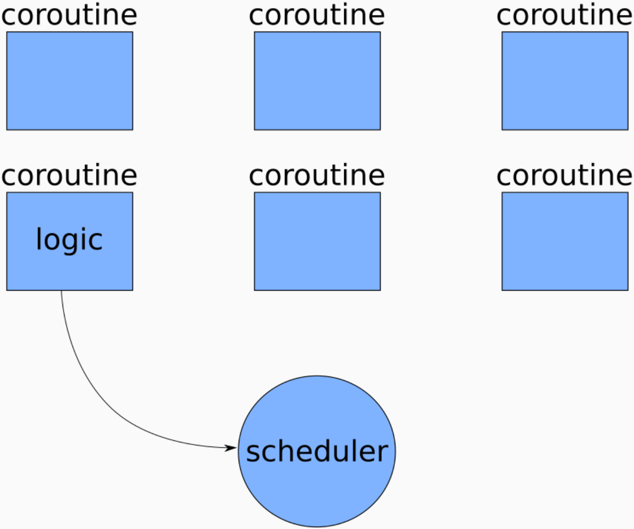
- lazy computation of the sequences (generators)
- possibility of introducing asynchronous code with one thread
- non blocking I/O operations

What are coroutines for?

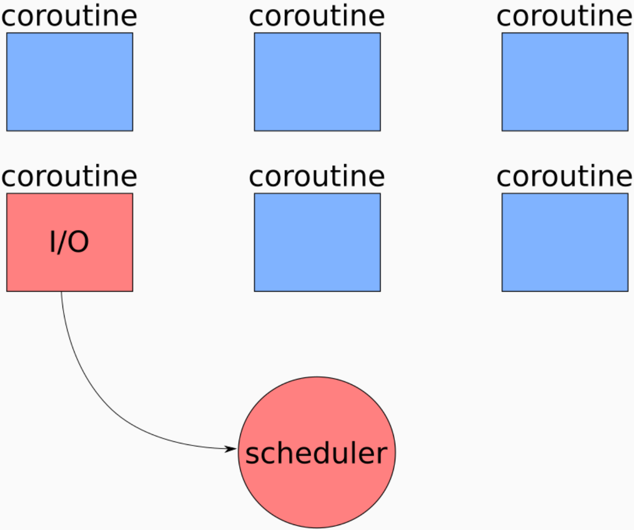
Common use cases for coroutines are:

- lazy computation of the sequences (generators)
- possibility of introducing asynchronous code with one thread
- non blocking I/O operations
- lightweight "concurrency"

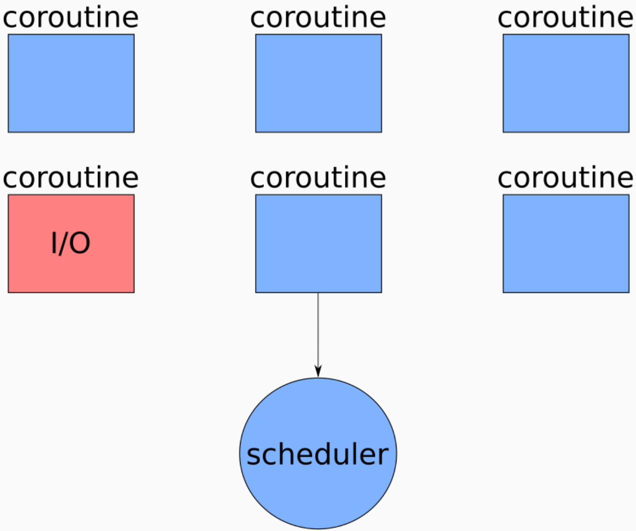
Example coroutines usage



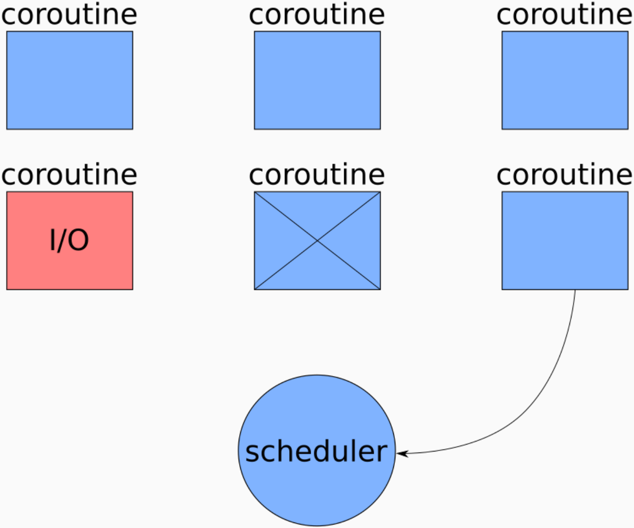
Example coroutines usage



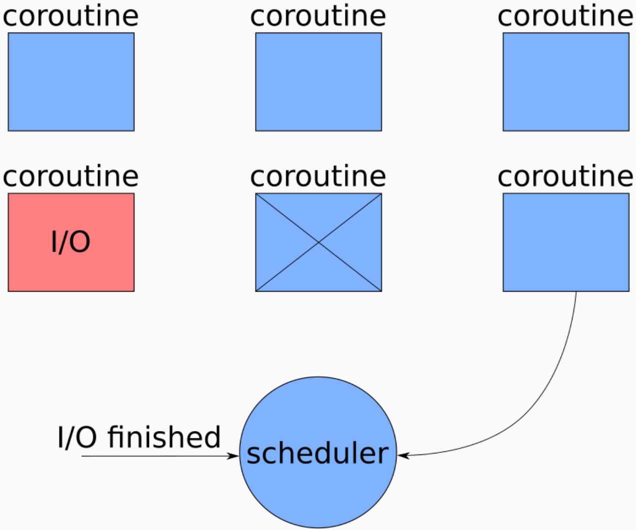
Example coroutines usage



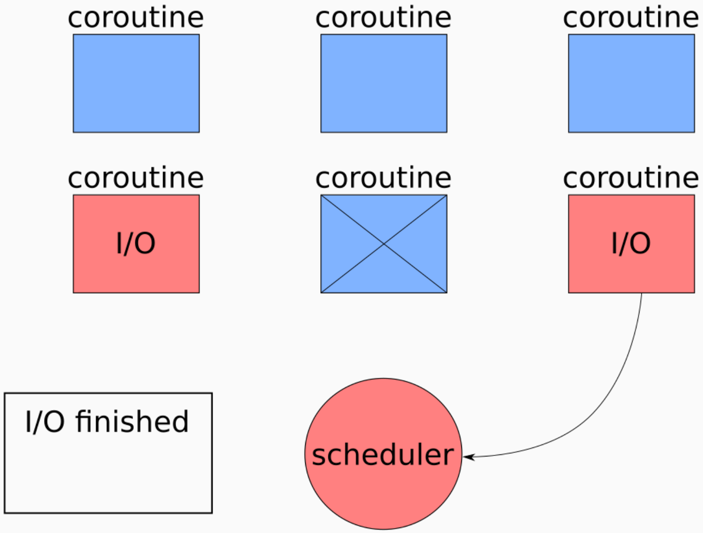
Example coroutines usage



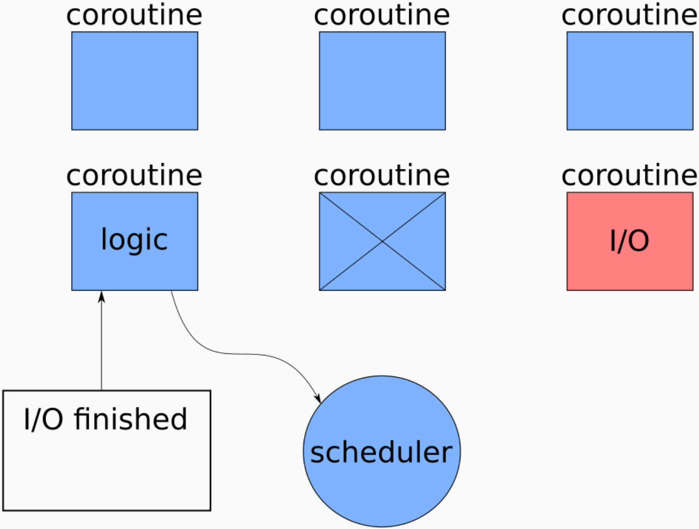
Example coroutines usage



Example coroutines usage



Example coroutines usage



The state

Imagine you want to suspend a function:

```
int foo(){
  int a = 1;

  co_yield a++;
  co_yield a++;
  co_yield a++;
}
```

The state

Imagine you want to suspend a function:

```
int foo(){
  int a = 1;

  co_yield a++;
  co_yield a++;
  co_yield a++;
}
```

issues:

- on suspension, variables need to be saved somewhere

The state

Imagine you want to suspend a function:

```
int foo(){
  int a = 1;

  co_yield a++;
  co_yield a++;
  co_yield a++;
}
```

issues:

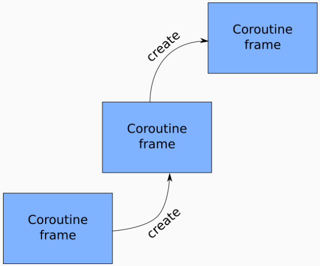
- on suspension, variables need to be saved somewhere
- coroutines needs to know where they suspended last time

How to achieve that?

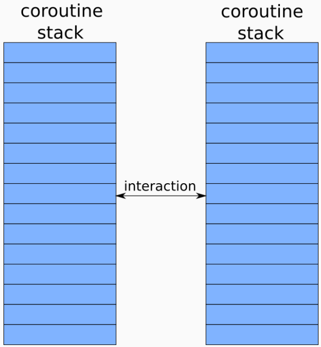
```
struct foo_state{
  int a=1;
  int recent_point=0;
};

int foo(foo_state* state){
  if(state.recent_point == 0) goto recent_point_0;
  if(state.recent_point == 1) goto recent_point_1;
  if(state.recent_point == 2) goto recent_point_2;

  recent_point_0:
  return state.a++;
  recent_point_1:
  return state.a++;
  recent_point_2:
  return state.a++;
}
```

Possible coroutines implementations

Language based



Library based



Closer look into Boost.Fiber

- Need to allocate a stack for the Fiber/Coroutine

Closer look into Boost.Fiber

- Need to allocate a stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below

Closer look into Boost.Fiber

- Need to allocate a stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance

Closer look into Boost.Fiber

- Need to allocate a stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- Hard to optimize by compilers

Closer look into built-in coroutines

Boost.Fiber

- Need to allocate a stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- Hard to optimize by compilers

built-in coroutines

- Need to allocate the frame for the Coroutine

Closer look into built-in coroutines

Boost.Fiber

- Need to allocate a stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- Hard to optimize by compilers

built-in coroutines

- Need to allocate the frame for the Coroutine
- Can be suspended only from the top level functions

Closer look into built-in coroutines

Boost.Fiber

- Need to allocate a stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- Hard to optimize by compilers

built-in coroutines

- Need to allocate the frame for the Coroutine
- Can be suspended only from the top level functions
- Minimal memory allocation

Closer look into built-in coroutines

Boost.Fiber

- Need to allocate a stack for the Fiber/Coroutine
- Can be suspended from the top level functions and below
- Allocation of the memory in advance
- Hard to optimize by compilers

built-in coroutines

- Need to allocate the frame for the Coroutine
- Can be suspended only from the top level functions
- Minimal memory allocation
- Easy to optimize by compilers

Coroutine declarations

Same as functions

```
// returned-type   name      arguments
//|-------------| |-------| |--------------|
   generator<int> fibonacci (int from_value);
```

Whether the function is a coroutine depends on its definition.

3 new keywords

`co_return`
: Returning (or not) value and finishing the coroutine

`co_yield`
: Returning intermediate value from the coroutine

`co_await`
: Awaiting completion of the "task"

Practical part I - using the cppcoro library

generators - Fibonacci sequence

```
cppcoro::generator<unsigned long long> fibonacci_gen() {
  std::array arr{0ull, 1ull};
  unsigned long long result=0;

  do {
    co_yield result;
    arr[0] = arr[1];
    arr[1] = result;
    result = arr[0] + arr[1];
  } while (result >= arr[1]);
}
```

generators - Fibonacci sequence

```cpp
cppcoro::generator<unsigned long long> fibonacci_gen() {
  std::array arr{0ull, 1ull};
  unsigned long long result=0;

  do {
    co_yield result;
    arr[0] = arr[1];
    arr[1] = result;
    result = arr[0] + arr[1];
  } while (result >= arr[1]);
}
```

generators - Fibonacci sequence

```cpp
cppcoro::generator<unsigned long long> fibonacci_gen() {
  std::array arr{0ull, 1ull};
  unsigned long long result=0;

  do {
    co_yield result;
    arr[0] = arr[1];
    arr[1] = result;
    result = arr[0] + arr[1];
  } while (result >= arr[1]);
}
```

generators - Fibonacci sequence

```
cppcoro::generator<unsigned long long> fibonacci_gen() {
  std::array arr{0ull, 1ull};
  unsigned long long result=0;

  do {
    co_yield result;
    arr[0] = arr[1];
    arr[1] = result;
    result = arr[0] + arr[1];
  } while (result >= arr[1]);
}
```

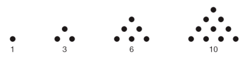
generators - Fibonacci sequence

```cpp
cppcoro::generator<unsigned long long> fibonacci_gen() {
  std::array arr{0ull, 1ull};
  unsigned long long result=0;

  do {
    co_yield result;
    arr[0] = arr[1];
    arr[1] = result;
    result = arr[0] + arr[1];
  } while (result >= arr[1]);
}
```

generators - exercise

Implement generator:

triangular number series



source

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

tasks and events

```
  single_consumer_event ev_ping;
  single_consumer_event ev_pong;
  sched(
    [&]() -> cppcoro::task<> {
      while (true) {
        co_await ev_ping;
        std::cout << "ping" << std::endl;
        ev_ping.reset();
        ev_pong.set();
      }
    }(),
    [&]() -> cppcoro::task<> {
      while (true) {
        ev_ping.set();
        co_await ev_pong;
        ev_pong.reset();
        std::cout << "pong" << std::endl;
        std::this_thread::sleep_for(1000ms);
      }
    }());
}
```

other (for now only msvc)

- mutexes
- file I/O operations
- networking operations

Theory - promise_types

promise_type

- promise_type is responsible for the coroutine's behavior:

promise_type

- promise_type is responsible for the coroutine's behavior:
    - for starting and stopping coroutines

promise_type

- promise_type is responsible for the coroutine's behavior:
  - for starting and stopping coroutines
  - for throwing unhandled exceptions

promise_type

- promise_type is responsible for the coroutine’s behavior:
    - for starting and stopping coroutines
    - for throwing unhandled exceptions
    - for handling the return value

promise_type

- promise_type is responsible for the coroutine’s behavior:
    - for starting and stopping coroutines
    - for throwing unhandled exceptions
    - for handling the return value
    - for handling yielded values

promise_type

- promise_type is responsible for the coroutine's behavior:
  - for starting and stopping coroutines
  - for throwing unhandled exceptions
  - for handling the return value
  - for handling yielded values
  - partially for waiting for the task's completion

promise_type

- promise_type is responsible for the coroutine’s behavior:
  - for starting and stopping coroutines
  - for throwing unhandled exceptions
  - for handling the return value
  - for handling yielded values
  - partially for waiting for the task’s completion
- promise_type is strongly connected with the coroutine’s returned type

promise_type

- promise_type is responsible for the coroutine's behavior:
  - for starting and stopping coroutines
  - for throwing unhandled exceptions
  - for handling the return value
  - for handling yielded values
  - partially for waiting for the task's completion
- promise_type is strongly connected with the coroutine's returned type
  - it can be a member of the returned type `returned_type::promise_type`

promise_type

- promise_type is responsible for the coroutine's behavior:
  - for starting and stopping coroutines
  - for throwing unhandled exceptions
  - for handling the return value
  - for handling yielded values
  - partially for waiting for the task's completion
- promise_type is strongly connected with the coroutine's returned type
  - it can be a member of the returned type `returned_type::promise_type`
  - it can be defined as `std::coroutine_traits<returned_type>::promise_type`

coroutine body

The compiler transform a coroutine's body into the following form:

```
{
  promise_type promise;
  auto&& return_object = promise.get_return_object();
  co_await promise.initial_suspend();

  try{
    //our coroutine_body
  }catch(...) {
    promise.unhandled_exception();
  }

  final_suspend:
  co_await promise.final_suspend();
}
```

coroutine body

The compiler transform a coroutine's body into the following form:

```
{
  promise_type promise;
  auto&& return_object = promise.get_return_object();
  co_await promise.initial_suspend();

  try{
    //our coroutine_body
  }catch(...) {
    promise.unhandled_exception();
  }

  final_suspend:
  co_await promise.final_suspend();
}
```

coroutine body

The compiler transform a coroutine's body into the following form:

```
{
  promise_type promise;
  auto&& return_object = promise.get_return_object();
  co_await promise.initial_suspend();

  try{
    //our coroutine_body
  }catch(...) {
    promise.unhandled_exception();
  }

  final_suspend:
  co_await promise.final_suspend();
}
```

coroutine body

The compiler transform a coroutine's body into the following form:

```
{
  promise_type promise;
  auto&& return_object = promise.get_return_object();
  co_await promise.initial_suspend();

  try{
    //our coroutine_body
  }catch(...) {
    promise.unhandled_exception();
  }

  final_suspend:
  co_await promise.final_suspend();
}
```

coroutine body

The compiler transform a coroutine's body into the following form:

```
{
  promise_type promise;
  auto&& return_object = promise.get_return_object();
  co_await promise.initial_suspend();

  try{
    //our coroutine_body
  }catch(...) {
    promise.unhandled_exception();
  }

  final_suspend:
  co_await promise.final_suspend();
}
```

coroutine body

The compiler transform a coroutine's body into the following form:

```
{
  promise_type promise;
  auto&& return_object = promise.get_return_object();
  co_await promise.initial_suspend();

  try{
    //our coroutine_body
  }catch(...) {
    promise.unhandled_exception();
  }

  final_suspend:
  co_await promise.final_suspend();
}
```

coroutine body

The compiler transform a coroutine’s body into the following form:

```
{
  promise_type promise;
  auto&& return_object = promise.get_return_object();
  co_await promise.initial_suspend();

  try{
    //our coroutine_body
  }catch(...) {
    promise.unhandled_exception();
  }

  final_suspend:
  co_await promise.final_suspend();
}
```

promise_type extensions

We can extend the coroutine body with:

- (mandatory) support for returning void or returning a value
- custom memory allocation algorithm (custom `operator new`)
- (optional) support for yielding intermediate values

co_return - support for returning from coroutine

`co_return`

Usage:

Translated to:

without expression:

```
co_return
```

with void expression:

```
co_return <void expression>
```

```
<expression>;
promise.return_void();
goto final_suspend;
```

co_return - support for returning from coroutine

`co_return`

Usage:

Translated to:

with non-void expression:

```
co_return <expression>
```

```
promise.return_value(<expression>);
goto final_suspend;
```

co_yield - support for returning intermediate values

co_yield

Usage:

```
co_yield <non-void expression>
```

Translated to:

```
co_await promise.
         yield_value(<expression>)
```

co_await shortly

`co_await std::experimental::suspend_always{} -`
suspends the coroutine

`co_await std::experimental::suspend_never{} -`
does nothing

Ok, ok but how do I even resume the coroutine?

The low-level interface to the type-erased coroutine is `coroutine_handle` object.

```
template<> struct coroutine_handle<void> {
  constexpr coroutine_handle() noexcept;
  constexpr coroutine_handle(nullptr_t) noexcept;
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr void* address() const noexcept;
  constexpr static coroutine_handle from_address(void* addr);

  constexpr explicit operator bool() const noexcept;
  bool done() const;

  void operator()() const;
  void resume() const;

  void destroy() const;
  /* ... */
};
```

Ok, ok but how do I even resume the coroutine?

The low-level interface to the type-erased coroutine is `coroutine_handle` object.

```
template<> struct coroutine_handle<void> {
  constexpr coroutine_handle() noexcept;
  constexpr coroutine_handle(nullptr_t) noexcept;
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr void* address() const noexcept;
  constexpr static coroutine_handle from_address(void* addr);

  constexpr explicit operator bool() const noexcept;
  bool done() const;

  void operator()() const;
  void resume() const;

  void destroy() const;
  /* ... */
};
```

Ok, ok but how do I even resume the coroutine?

The low-level interface to the type-erased coroutine is `coroutine_handle` object.

```
template<> struct coroutine_handle<void> {
  constexpr coroutine_handle() noexcept;
  constexpr coroutine_handle(nullptr_t) noexcept;
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr void* address() const noexcept;
  constexpr static coroutine_handle from_address(void* addr);

  constexpr explicit operator bool() const noexcept;
  bool done() const;

  void operator()() const;
  void resume() const;

  void destroy() const;
  /* ... */
};
```

Ok, ok but how do I even resume the coroutine?

The low-level interface to the type-erased coroutine is `coroutine_handle` object.

```
template<> struct coroutine_handle<void> {
  constexpr coroutine_handle() noexcept;
  constexpr coroutine_handle(nullptr_t) noexcept;
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr void* address() const noexcept;
  constexpr static coroutine_handle from_address(void* addr);

  constexpr explicit operator bool() const noexcept;
  bool done() const;

  void operator()() const;
  void resume() const;

  void destroy() const;
  /* ... */
};
```

Ok, ok but how do I even resume the coroutine?

The low-level interface to the type-erased coroutine is `coroutine_handle` object.

```
template<> struct coroutine_handle<void> {
  constexpr coroutine_handle() noexcept;
  constexpr coroutine_handle(nullptr_t) noexcept;
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr void* address() const noexcept;
  constexpr static coroutine_handle from_address(void* addr);

  constexpr explicit operator bool() const noexcept;
  bool done() const;

  void operator()() const;
  void resume() const;

  void destroy() const;
  /* ... */
};
```

Ok, ok but how do I even resume the coroutine?

The low-level interface to the type-erased coroutine is `coroutine_handle` object.

```
template<> struct coroutine_handle<void> {
  constexpr coroutine_handle() noexcept;
  constexpr coroutine_handle(nullptr_t) noexcept;
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr void* address() const noexcept;
  constexpr static coroutine_handle from_address(void* addr);

  constexpr explicit operator bool() const noexcept;
  bool done() const;

  void operator()() const;
  void resume() const;

  void destroy() const;
  /* ... */
};
```

Ok, ok but how do I even resume the coroutine?

The low-level interface to the type-erased coroutine is `coroutine_handle` object.

```
template<> struct coroutine_handle<void> {
  constexpr coroutine_handle() noexcept;
  constexpr coroutine_handle(nullptr_t) noexcept;
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr void* address() const noexcept;
  constexpr static coroutine_handle from_address(void* addr);

  constexpr explicit operator bool() const noexcept;
  bool done() const;

  void operator()() const;
  void resume() const;

  void destroy() const;
  /* ... */
};
```

And how do I get the coroutine_handle object?

coroutine_handles are specialized for promise_type

```
template<class Promise>
struct coroutine_handle : coroutine_handle<>
{
  using coroutine_handle<>::coroutine_handle;
  static coroutine_handle from_promise(Promise&);
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr static coroutine_handle from_address(void* addr);

  Promise& promise() const;
};
```

And how do I get the coroutine_handle object?

coroutine_handles are specialized for promise_type

```
template<class Promise>
struct coroutine_handle : coroutine_handle<>
{
  using coroutine_handle<>::coroutine_handle;
  static coroutine_handle from_promise(Promise&);
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr static coroutine_handle from_address(void* addr);

  Promise& promise() const;
};
```

And how do I get the coroutine_handle object?

coroutine_handles are specialized for promise_type

```
template<class Promise>
struct coroutine_handle : coroutine_handle<>
{
  using coroutine_handle<>::coroutine_handle;
  static coroutine_handle from_promise(Promise&);
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr static coroutine_handle from_address(void* addr);

  Promise& promise() const;
};
```

And how do I get the coroutine_handle object?

coroutine_handles are specialized for promise_type

```
template<class Promise>
struct coroutine_handle : coroutine_handle<>
{
  using coroutine_handle<>::coroutine_handle;
  static coroutine_handle from_promise(Promise&);
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr static coroutine_handle from_address(void* addr);

  Promise& promise() const;
};
```

And how do I get the coroutine_handle object?

coroutine_handles are specialized for promise_type

```
template<class Promise>
struct coroutine_handle : coroutine_handle<>
{
  using coroutine_handle<>::coroutine_handle;
  static coroutine_handle from_promise(Promise&);
  coroutine_handle& operator=(nullptr_t) noexcept;

  constexpr static coroutine_handle from_address(void* addr);

  Promise& promise() const;
};
```

Promise_type exercise

promise_type exercise - lazy

Type for lazy initialization

Requirements:

- synchronous (no support for multithreading + no support for co_await).
- no sharing of the value (no copy, only move constructor).
- interface similar to std::optional.

promise_type exercise - generator

Type for generating sequences

Requirements:

- synchronous (no support for multithreading + no support for co_await).
- next method should return the value and resume the coroutine.
- interface similar to std::optional

Asynchronous coroutines

co_await

co_await

- represents awaiting the completion of operations
- its argument is (usually) called awaitable
- its result is usually called awaiter

co_await translation

If the compiler meets a co_await it translates it into the following code:

```
{
  std::exception_ptr exception = nullptr;
  if (not a.await_ready()) {
    suspend_coroutine();

    <await_suspend>
  }

  resume_point:
  if(exception)
    std::rethrow_exception(exception);
  /*return*/ a.await_resume();
}
```

co_await translation

If the compiler meets a co_await it translates it into the following code:

```
{
  std::exception_ptr exception = nullptr;
  if (not a.await_ready()) {
    suspend_coroutine();

    <await_suspend>
  }

  resume_point:
  if(exception)
    std::rethrow_exception(exception);
  /*return*/ a.await_resume();
}
```

co_await translation

If the compiler meets a co_await it translates it into the following code:

```
{
  std::exception_ptr exception = nullptr;
  if (not a.await_ready()) {
    suspend_coroutine();

    <await_suspend>
  }

  resume_point:
  if(exception)
    std::rethrow_exception(exception);
  /*return*/ a.await_resume();
}
```

co_await translation

If the compiler meets a co_await it translates it into the following code:

```
{
  std::exception_ptr exception = nullptr;
  if (not a.await_ready()) {
    suspend_coroutine();

    <await_suspend>
  }

  resume_point:
  if(exception)
    std::rethrow_exception(exception);
  /*return*/ a.await_resume();
}
```

co_await translation

If the compiler meets a co_await it translates it into the following code:

```
{
  std::exception_ptr exception = nullptr;
  if (not a.await_ready()) {
    suspend_coroutine();

    <await_suspend>
  }

  resume_point:
  if(exception)
    std::rethrow_exception(exception);
  /*return*/ a.await_resume();
}
```

co_await translation

If the compiler meets a co_await it translates it into the following code:

```
{
  std::exception_ptr exception = nullptr;
  if (not a.await_ready()) {
    suspend_coroutine();

    <await_suspend>
  }

  resume_point:
  if(exception)
    std::rethrow_exception(exception);
  /*return*/ a.await_resume();
}
```

Await suspend

await suspend has following form:

```
awaitable.await_suspend(this_coroutine_handle);
```

`await_suspend` might return:

- `void`

```
//if await_suspend returns void
try {
  a.await_suspend(coroutine_handle);
  return_to_the_caller();
} catch (...) {
  exception = std::current_exception();
  goto resume_point;
}
//endif
```

Await suspend

await suspend has following form:

```
awaitable.await_suspend(this_coroutine_handle);
```

`await_suspend` might return:

- `void`
- `bool`

```
//if await_suspend returns bool
bool await_suspend_result;
try {
  await_suspend_result = a.await_suspend(
                            coroutine_handle);
} catch (...) {/*...*/}

if (not await_suspend_result)
  goto resume_point;
return_to_the_caller();
//endif
```

Await suspend

await suspend has following form:

```
awaitable.await_suspend(this_coroutine_handle);
```

`await_suspend` might return:

- `void`
- `bool`
- `coroutine_handle`

```
//if await_suspend returns coroutine_handle
decltype(a.await_suspend(
  std::declval<coro_handle_t>()))
another_coro_handle;
try {
  another_coro_handle = a.await_suspend(
                             coroutine_handle);
} catch (...) {/*...*/}

another_coro_handle.resume();
return_to_the_caller();
//endif
```

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform

is performed only if promise has await_transform function declared

```
co_await promise.await_transform(<expr>);
```

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform
- acquiring awaiter

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform
- acquiring awaiter
  - co_await operator

is performed only if awaitable has co_await operator

```
auto&& awaiter =
      <awaitable>.operator co_await();
```

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform
- acquiring awaiter
  - co_await operator
  - global co_await operator

is performed only if awaitable there is matching global co_await operator

```
auto&& awaiter =
      operator co_await(<awaitable>);
```

Awaitable and Awaiter

Awaitable is an object, which is an operand of the co_await operator

co_await <expression> expression will be processed in following manner

- await_transform
- acquiring awaiter
  - co_await operator
  - global co_await operator
  - awaitable to awaiter

```
auto&& awaiter = <awaitable>;
```

Asynchronous coroutines - exercises

What is the task?

```cpp
cppcoro::task<int> count_lines(std::string path)
{
  auto file = co_await cppcoro::read_only_file::open(path);

  int lineCount = 0;
  char buffer[1024];
  size_t bytesRead;
  std::uint64_t offset = 0;
  do
  {
    bytesRead = co_await file.read(offset, buffer, sizeof(buffer));
    lineCount += std::count(buffer, buffer + bytesRead, '\n');
    offset += bytesRead;
  } while (bytesRead > 0);

  co_return lineCount;
}
```

What is the task?

```
cppcoro::task<int> count_lines(std::string path)
{
  auto file = co_await cppcoro::read_only_file::open(path);

  int lineCount = 0;
  char buffer[1024];
  size_t bytesRead;
  std::uint64_t offset = 0;
  do
  {
    bytesRead = co_await file.read(offset, buffer, sizeof(buffer));
    lineCount += std::count(buffer, buffer + bytesRead, '\n');
    offset += bytesRead;
  } while (bytesRead > 0);

  co_return lineCount;
}
```

What is the task?

```cpp
cppcoro::task<int> count_lines(std::string path)
{
  auto file = co_await cppcoro::read_only_file::open(path);

  int lineCount = 0;
  char buffer[1024];
  size_t bytesRead;
  std::uint64_t offset = 0;
  do
  {
    bytesRead = co_await file.read(offset, buffer, sizeof(buffer));
    lineCount += std::count(buffer, buffer + bytesRead, '\n');
    offset += bytesRead;
  } while (bytesRead > 0);

  co_return lineCount;
}
```

What is the task?

```cpp
cppcoro::task<int> count_lines(std::string path)
{
  auto file = co_await cppcoro::read_only_file::open(path);

  int lineCount = 0;
  char buffer[1024];
  size_t bytesRead;
  std::uint64_t offset = 0;
  do
  {
    bytesRead = co_await file.read(offset, buffer, sizeof(buffer));
    lineCount += std::count(buffer, buffer + bytesRead, '\n');
    offset += bytesRead;
  } while (bytesRead > 0);

  co_return lineCount;
}
```

What is the task?

```
cppcoro::task<int> count_lines(std::string path)
{
  auto file = co_await cppcoro::read_only_file::open(path);

  int lineCount = 0;
  char buffer[1024];
  size_t bytesRead;
  std::uint64_t offset = 0;
  do
  {
    bytesRead = co_await file.read(offset, buffer, sizeof(buffer));
    lineCount += std::count(buffer, buffer + bytesRead, '\n');
    offset += bytesRead;
  } while (bytesRead > 0);

  co_return lineCount;
}
```

What is the task?

```cpp
cppcoro::task<int> count_lines(std::string path)
{
  auto file = co_await cppcoro::read_only_file::open(path);

  int lineCount = 0;
  char buffer[1024];
  size_t bytesRead;
  std::uint64_t offset = 0;
  do
  {
    bytesRead = co_await file.read(offset, buffer, sizeof(buffer));
    lineCount += std::count(buffer, buffer + bytesRead, '\n');
    offset += bytesRead;
  } while (bytesRead > 0);

  co_return lineCount;
}
```

task

Type for asynchronous operations

Requirements:

- single-threaded
- asynchronous (support for the co_await)
- coroutine after finishing must resume the co_awaiting coroutine
- some kind of the executor needed to start the coroutines

event

Type for communication of the tasks

Requirements:

- stores information whether the event is set.
- stores the continuation object.
- launches the continuation on setting up the event.

Thank you!

Questions?

recommended lectures:

- Lewiss Baker blog
- My blog
- "programista" magazine
- current C++ standard draft
- coroutine channel on cpplang slack

recommended videos:

- Gor Nishanov
  "C++ Coroutines: Under the covers"
- Toby Allsopp
  "Coroutines: what can't they do"
- James McNellis
  "Introduction to C++ Coroutines"