Generators

Introduction to the coroutines.

Dawid Pilarski

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

Agenda

Coroutine theory - what are the coroutines?

Practical part I - using cppcoro library

Theory - promise_types

Promise_type excercise

Coroutine theory - what are the coroutines?

What are the coroutines?

Coroutines are generalization of the function, that can be:

- created

What are the coroutines?

Coroutines are generalization of the function, that can be:

- created
- called

What are the coroutines?

Coroutines are generalization of the function, that can be:

- created
- called
- returned from

What are the coroutines?

Coroutines are generalization of the function, that can be:

- created
- called
- returned from
- suspended

What are the coroutines?

Coroutines are generalization of the function, that can be:

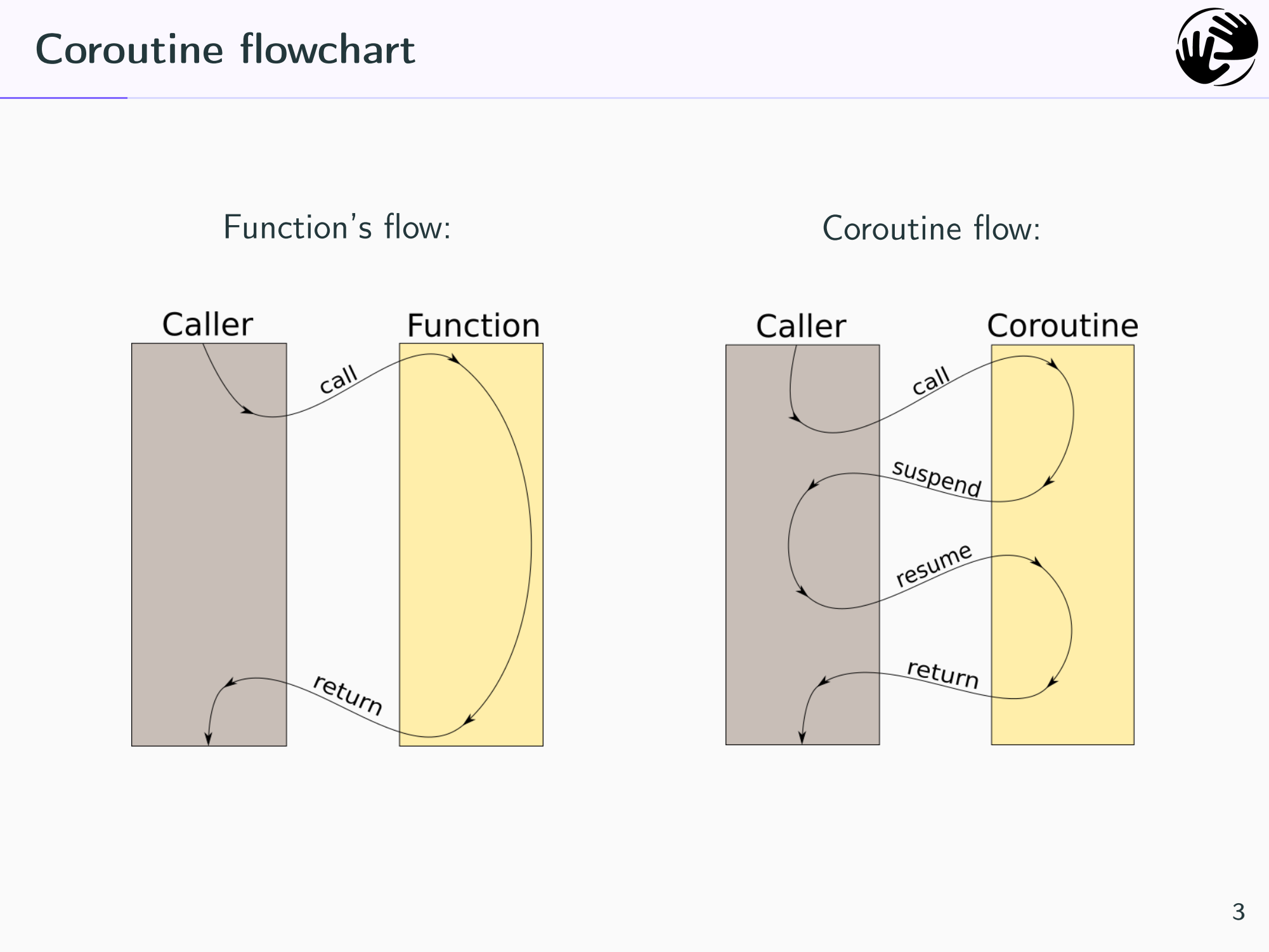
- created
- called
- returned from
- suspended
- resumed

What are the 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 the coroutines are:

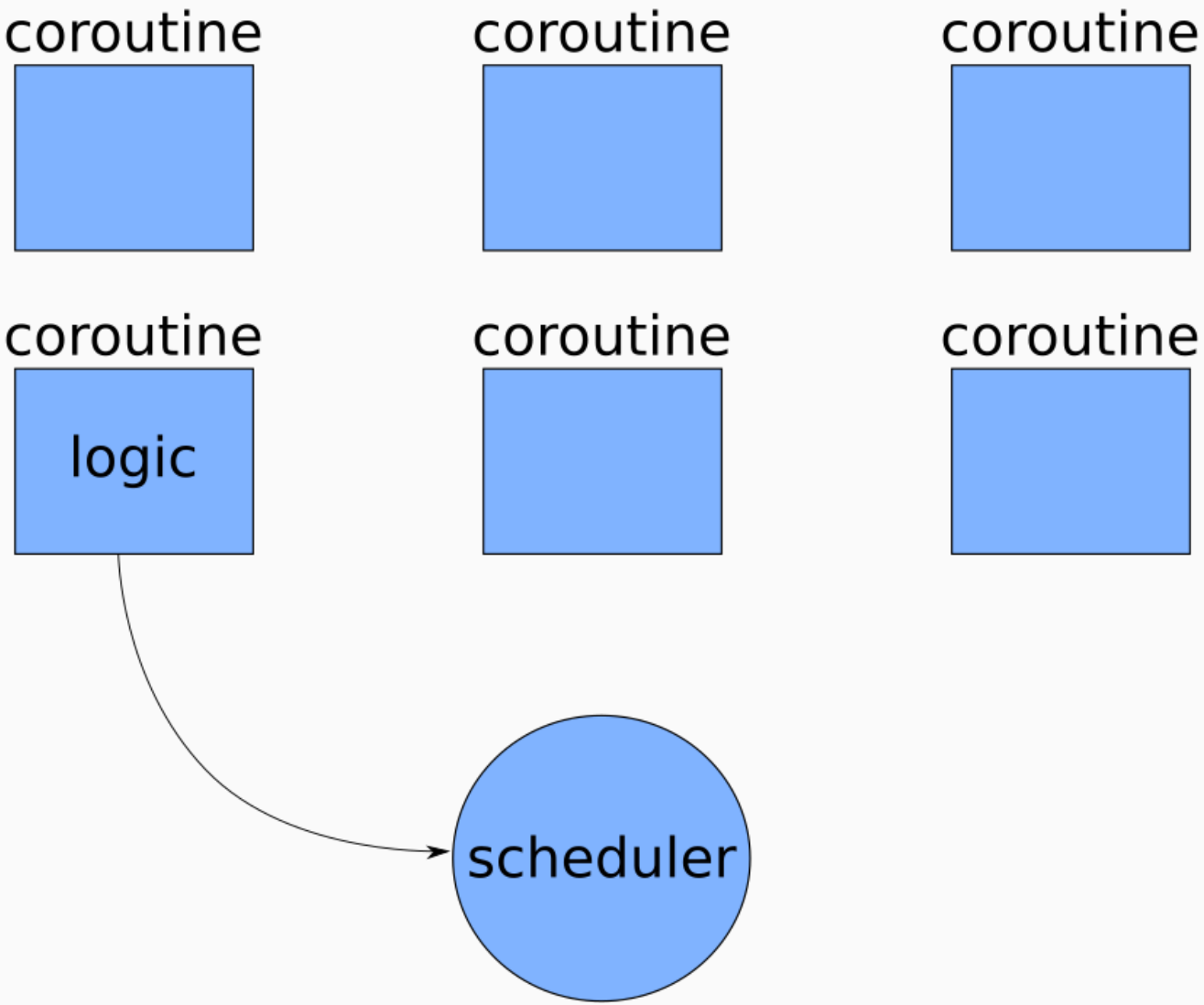
- lazy computation of the sequences (generators)

What are coroutines for?

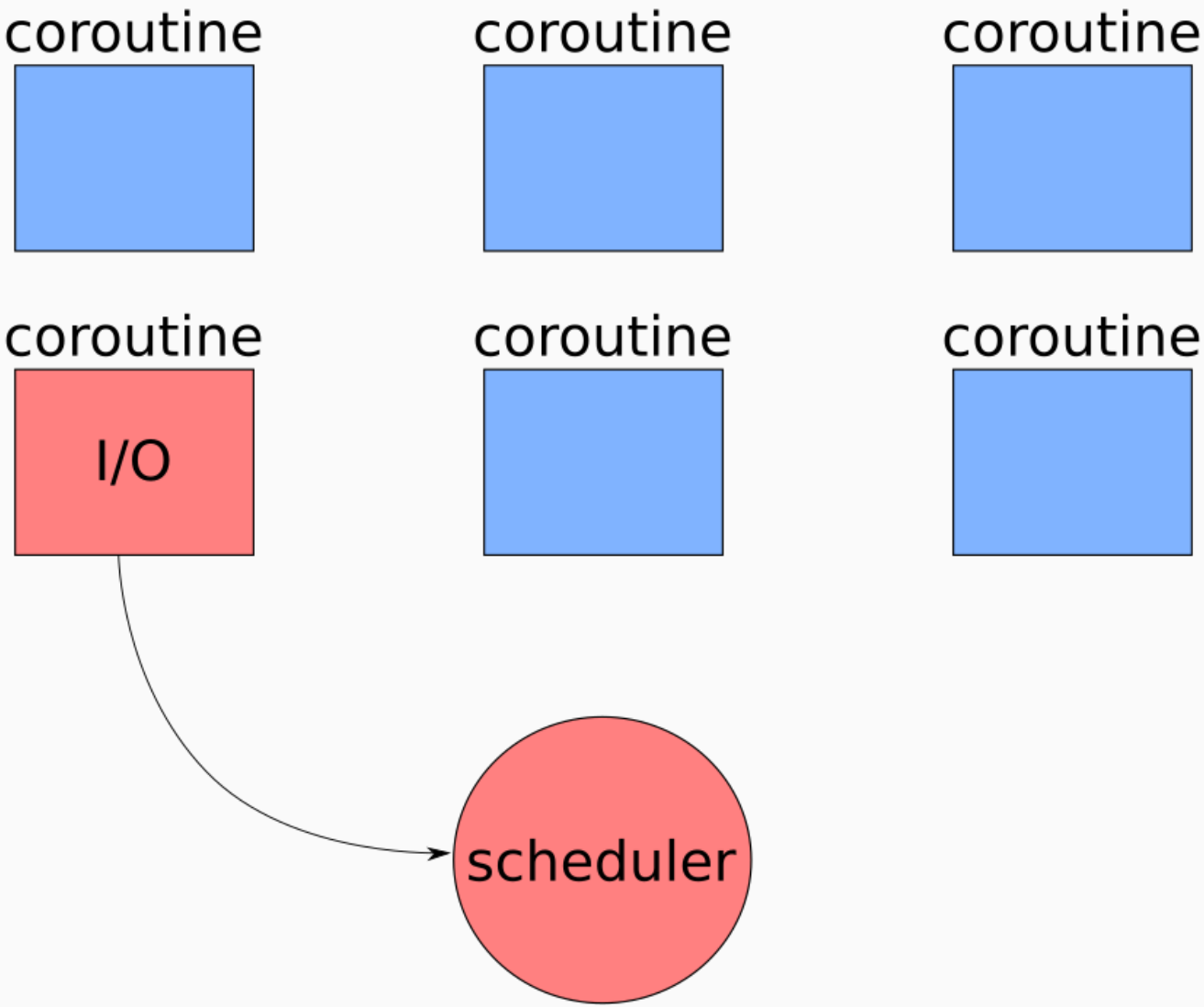
Common use cases for the coroutines are:

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

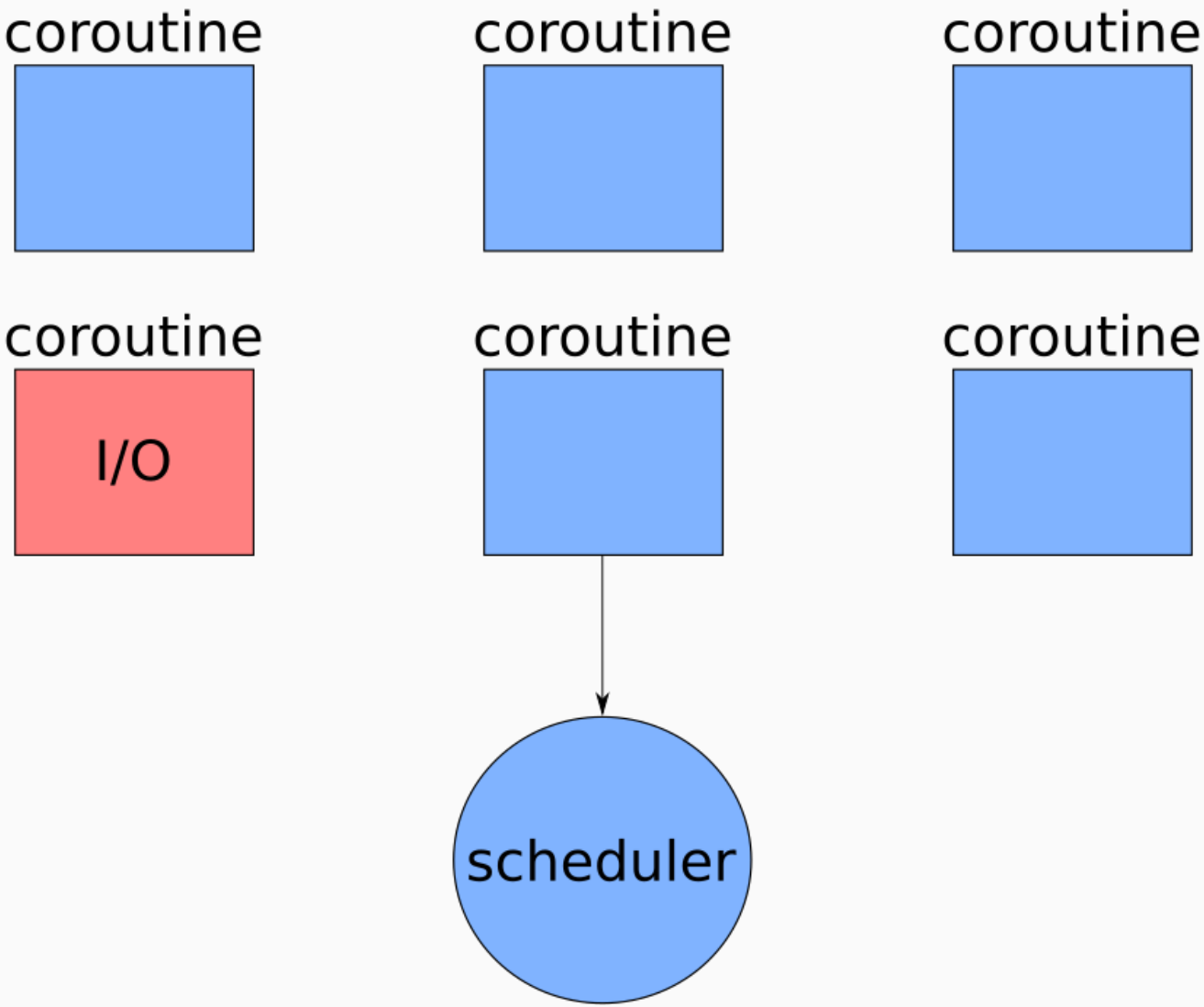
Example coroutines usage



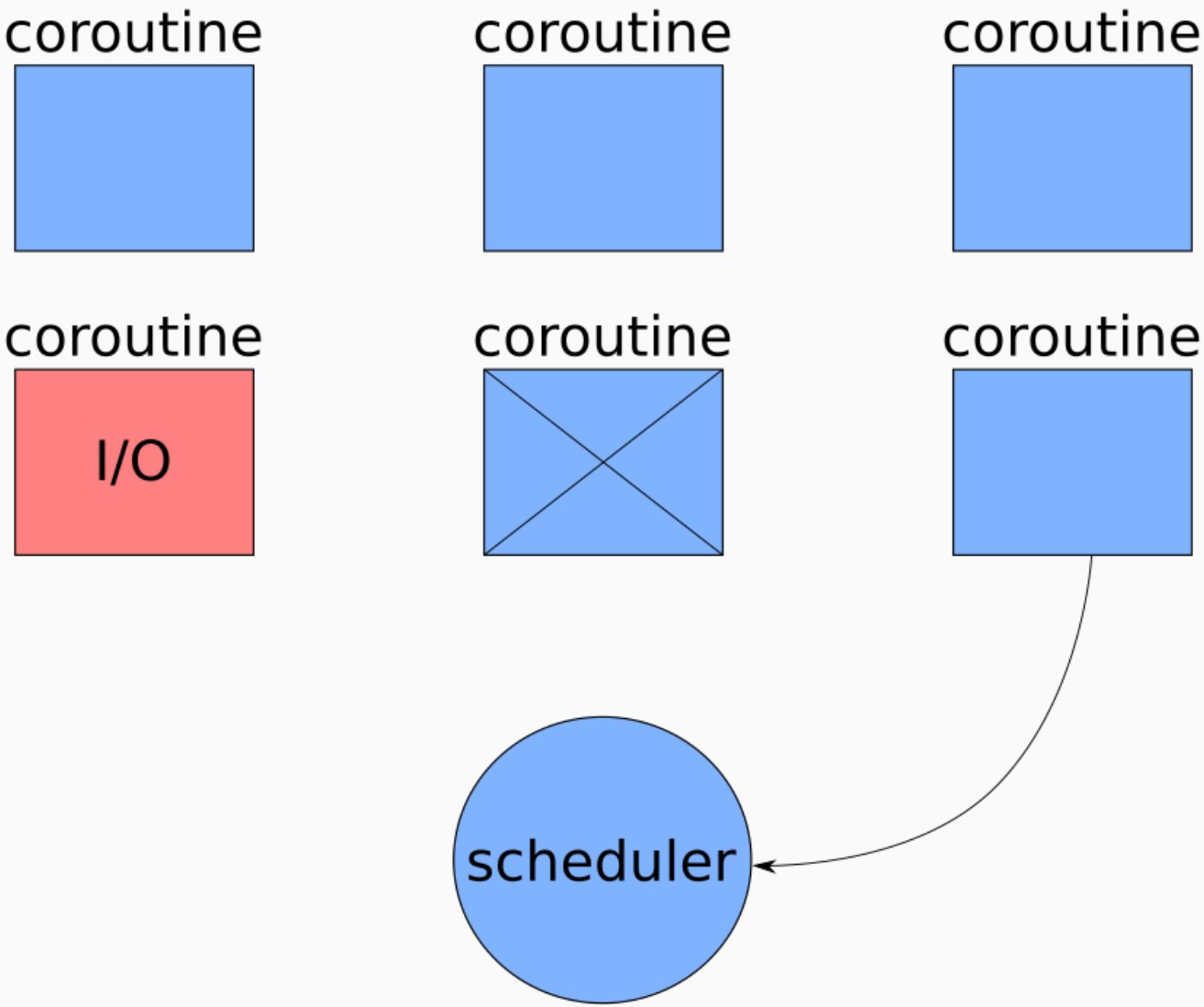
Example coroutines usage



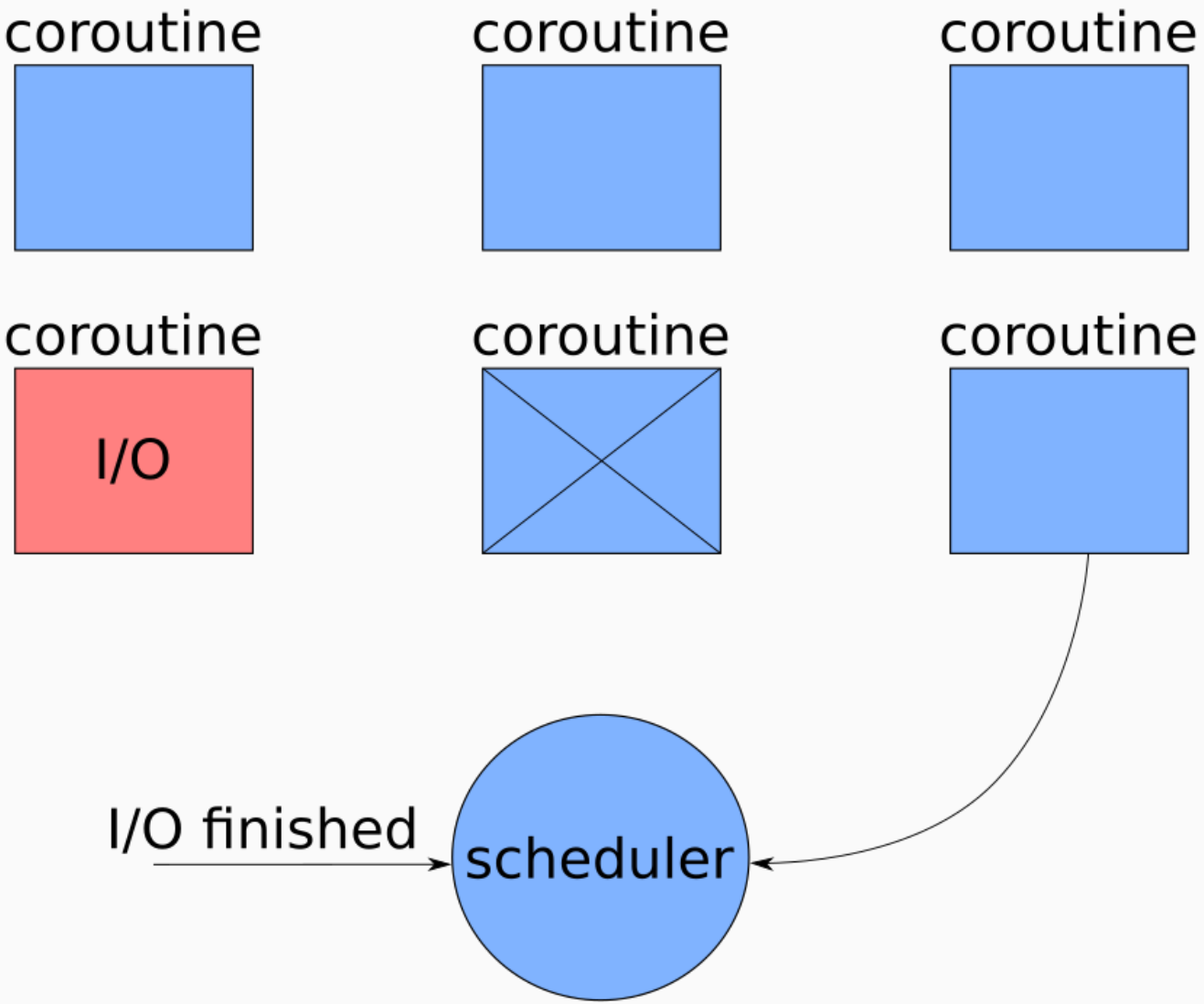
Example coroutines usage



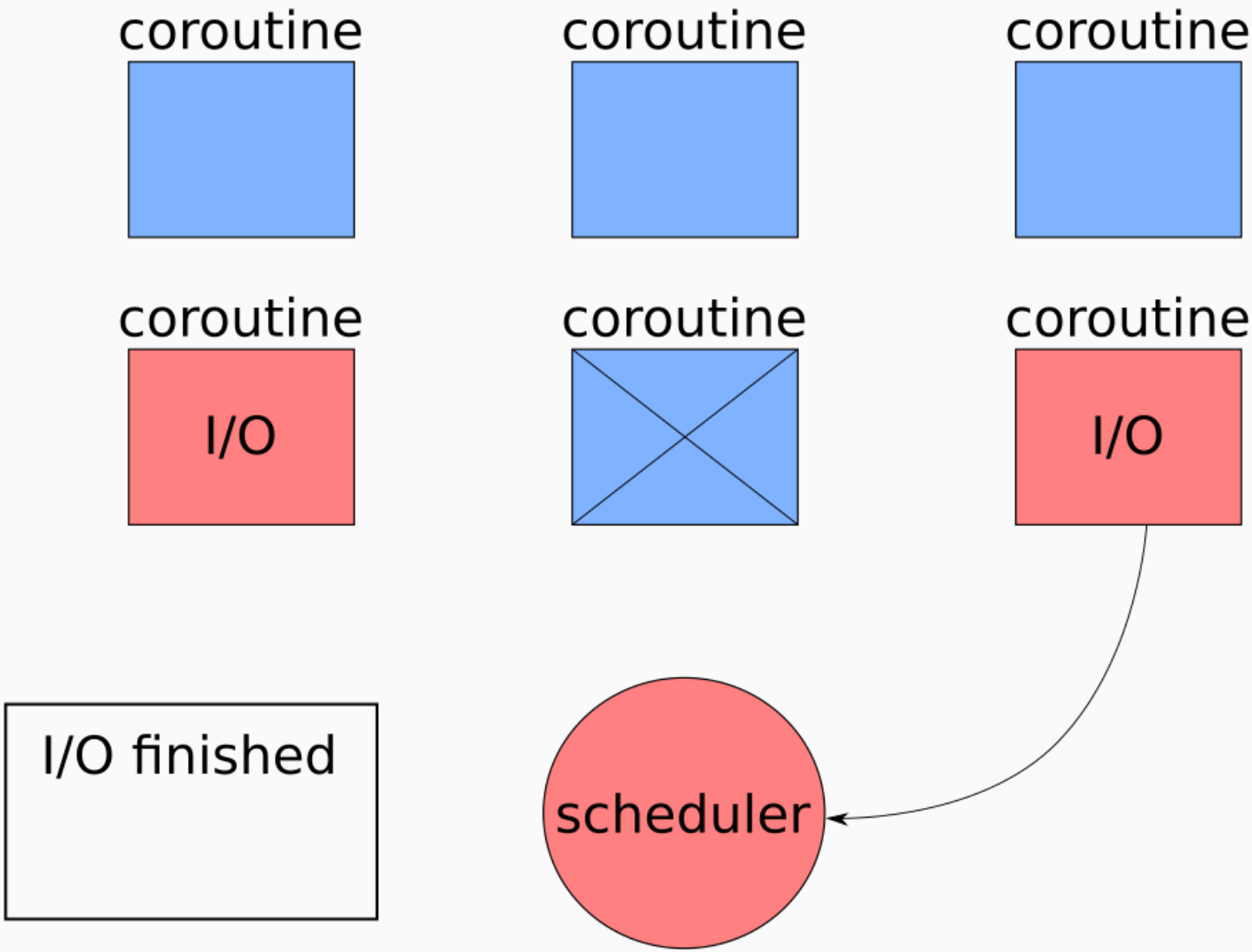
Example coroutines usage



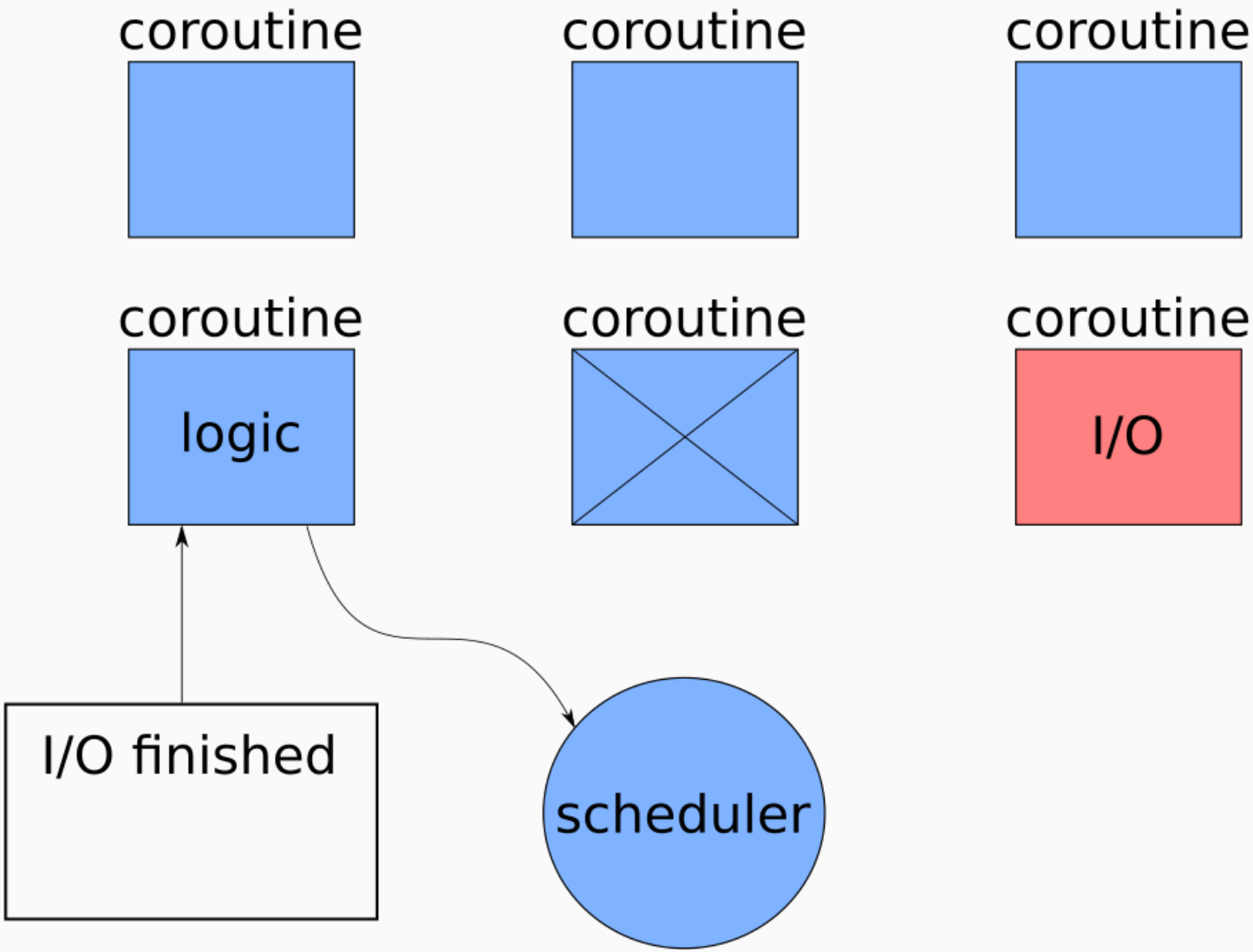
Example coroutines usage



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 needs 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 needs to be saved somewhere
- coroutine needs to know where it suspended last time

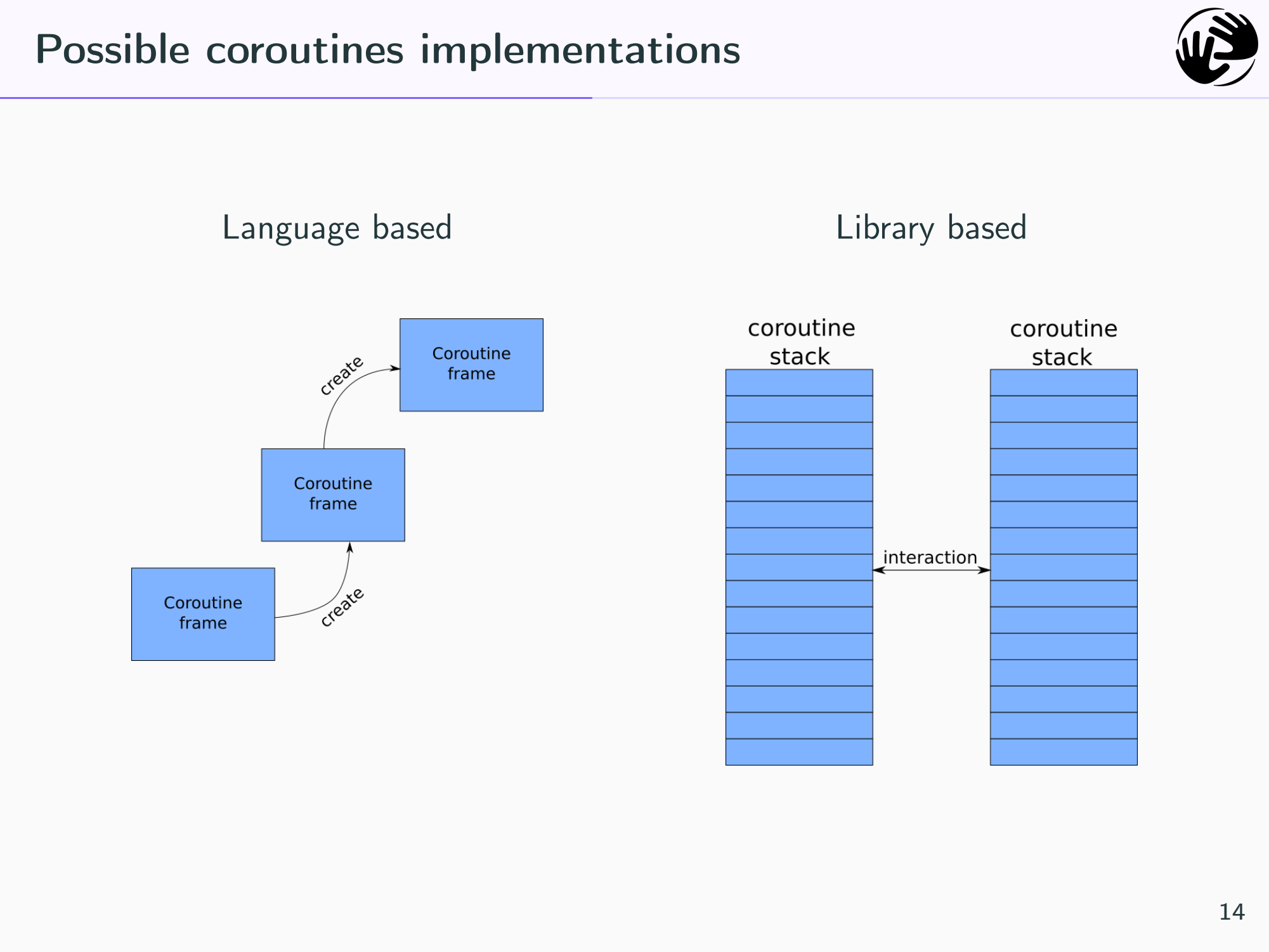
How to achieve that?

```
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 the stack for the Fiber/Coroutine

Closer look into Boost.Fiber

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

Closer look into Boost.Fiber

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

Closer look into built-in coroutines

Boost.Fiber

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

built-in coroutines

- Need to allocate the frame for the Coroutine

Closer look into built-in coroutines

Boost.Fiber

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

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

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

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 it's 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 cppcoro library

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

```
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

```
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]);
}
```

other (for now only msvc)

- mutexes
- file I/O operations
- networking operations

Theory - promise_types

promise_type

- promise_type is responsible for coroutine's behavior:

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start and stop

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start and stop
  - on throwing unhandled exception

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start and stop
  - on throwing unhandled exception
  - on returning the value

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start and stop
  - on throwing unhandled exception
  - on returning the value
  - on yielding the value

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start and stop
  - on throwing unhandled exception
  - on returning the value
  - on yielding the value
  - partially on waiting for the task's completion

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start and stop
  - on throwing unhandled exception
  - on returning the value
  - on yielding the value
  - partially on waiting for the task's completion
- promise_type is strongly connected with coroutine's returned type

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start and stop
  - on throwing unhandled exception
  - on returning the value
  - on yielding the value
  - partially on waiting for the task's completion
- promise_type is strongly connected with coroutine's returned type
  - it can be a member of the returned type `returned_type::promise_type`

promise_type

- promise_type is responsible for coroutine's behavior:
  - on coroutine's start and stop
  - on throwing unhandled exception
  - on returning the value
  - on yielding the value
  - partially on waiting for the task's completion
- promise_type is strongly connected with 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

Each time we write coroutine, compiler modifies it's body into 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

Each time we write coroutine, compiler modifies it's body into 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

Each time we write coroutine, compiler modifies it's body into 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

Each time we write coroutine, compiler modifies it's body into 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

Each time we write coroutine, compiler modifies it's body into 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

Each time we write coroutine, compiler modifies it's body into 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

Each time we write coroutine, compiler modifies it's body into 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 value
- custom memory allocation algorithm (custom `operator new`)
- (optional) support for yielding intermediate values

co_return - support for returning from coroutine

`co_return`

Usage:

without expression:

```
co_return
```

with void expression:

```
co_return <void expression>
```

Translated to:

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

co_return - support for returning from coroutine

`co_return`

Usage:

with non-void expression:

```
co_return <expression>
```

Translated to:

```
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 excercise

promise_type excercise - 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.

Thank you!

Questions?

recommended lecture:

- 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"