Presentation Material

C++ Coroutines from scratch

OOP
Online
2023-02-08

Andreas Fertig

Write unique code!

All programs, procedures and electronic circuits contained in this book have been created to the best of our knowledge and belief and have been tested with care. Nevertheless, errors cannot be completely ruled out. For this reason, the program material contained in this book is not associated with any obligation or guarantee of any kind. The author therefore assumes no responsibility and will not accept any liability, consequential or otherwise, arising in any way from the use of this program material or parts thereof.

Version: v1.0

The reproduction of common names, trade names, product designations, etc. in this work does not justify the assumption that such names are to be regarded as free in the sense of trademark and brand protection legislation and can therefore be used by anyone, even without special identification.

Planning, typesetting and cover design: Andreas Fertig

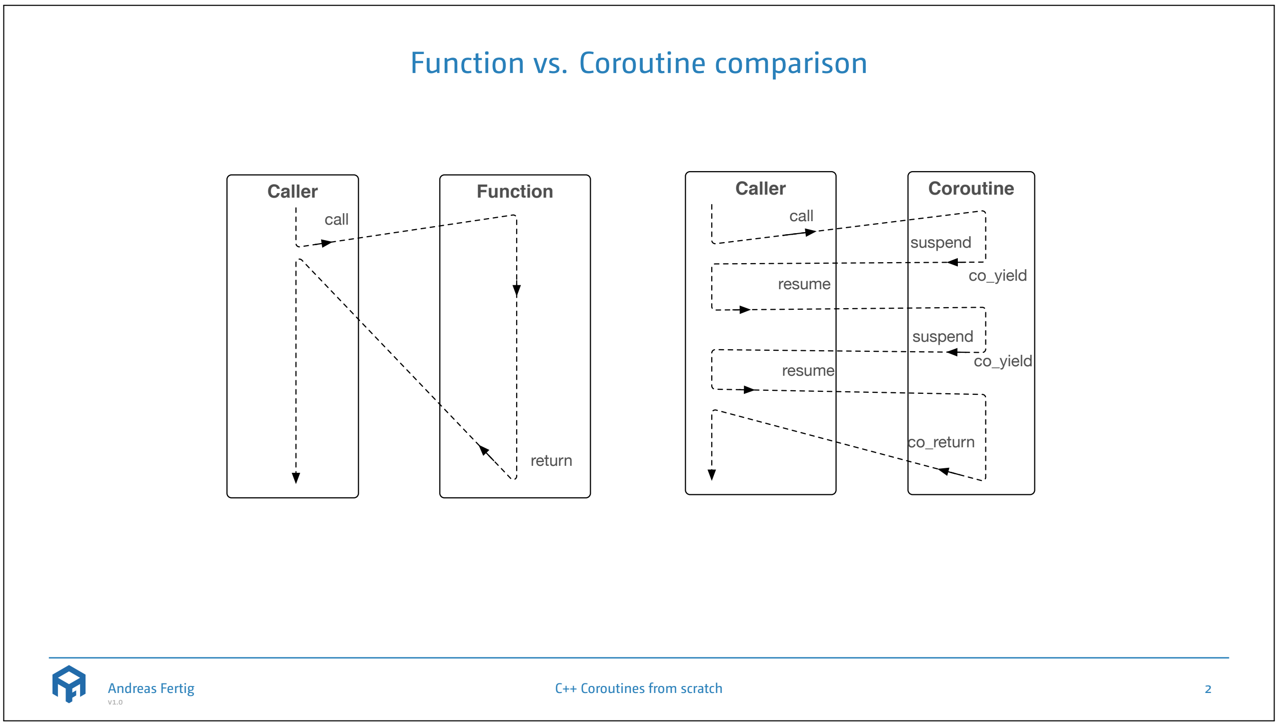
Production and publishing: Andreas Fertig

Style and conventions

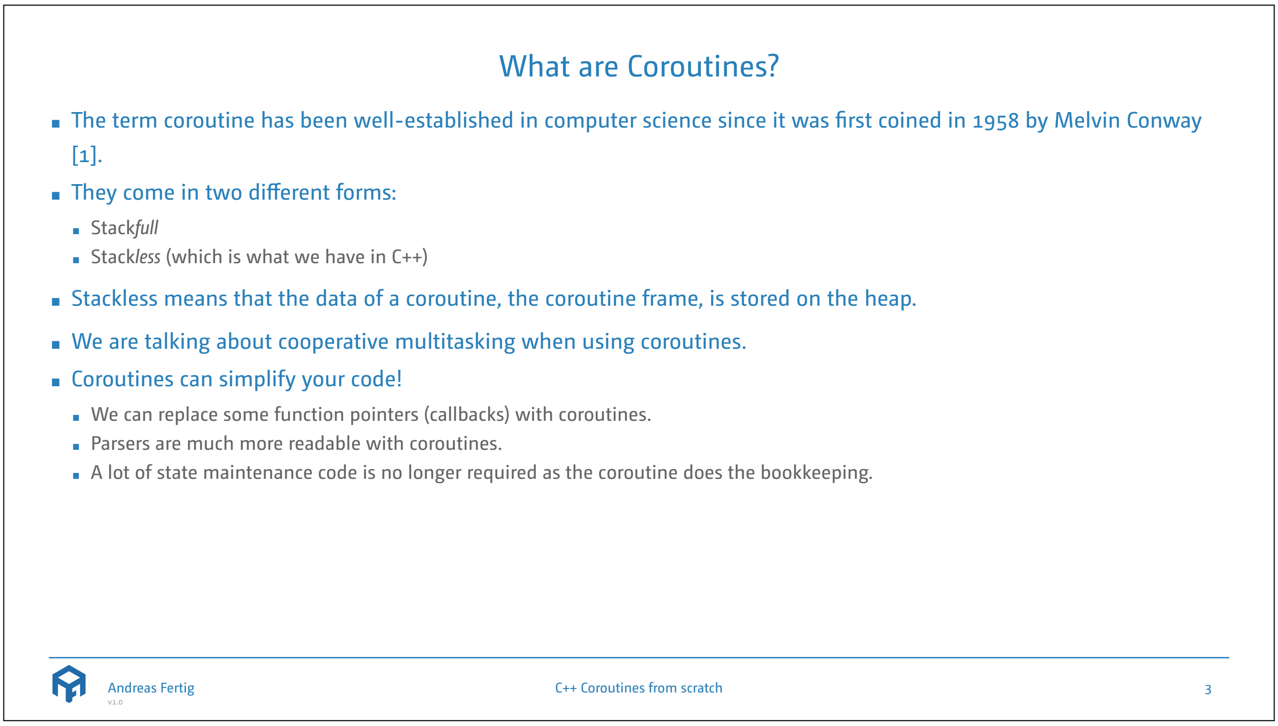
The following shows the execution of a program. I used the Linux way here and skipped supplying the desired output name, resulting in a.out as the program name.

```
$ ./a.out
Hello, C++!
```

- <string> stands for a header file with the name string
- [[xyz]] marks a C++ attribute with the name xyz.

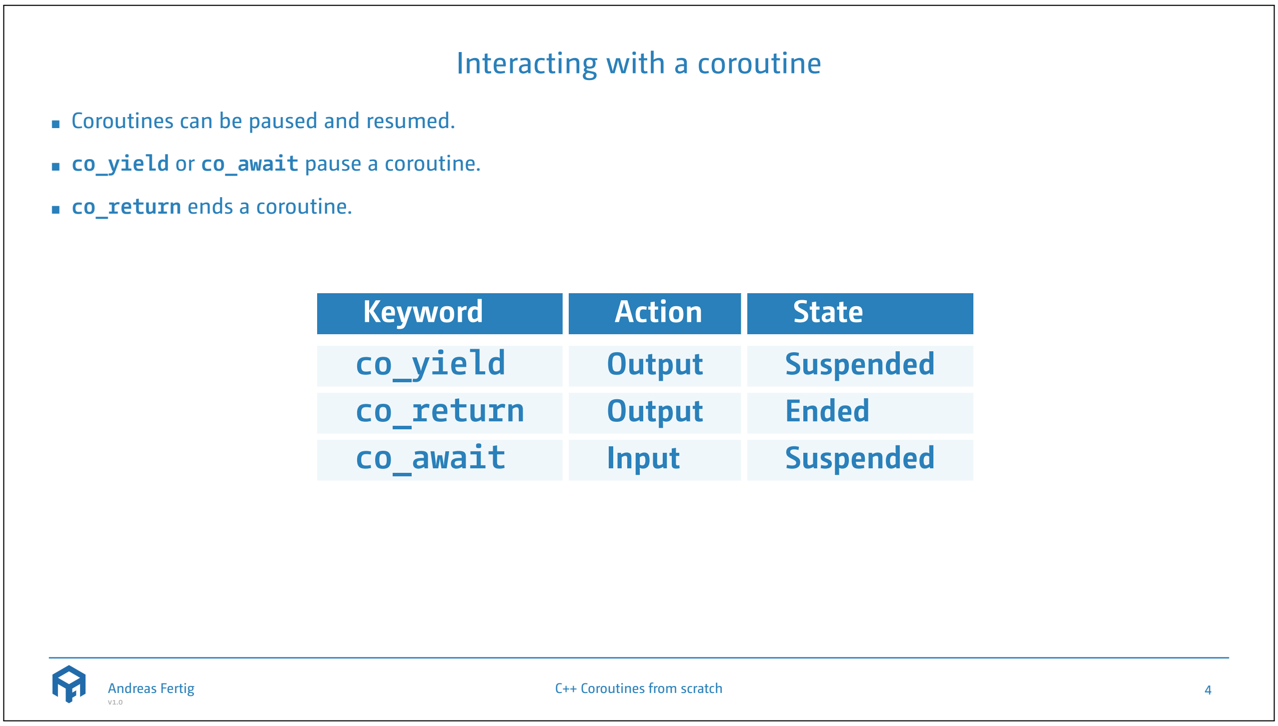


Function vs. Coroutine comparison



What are Coroutines?

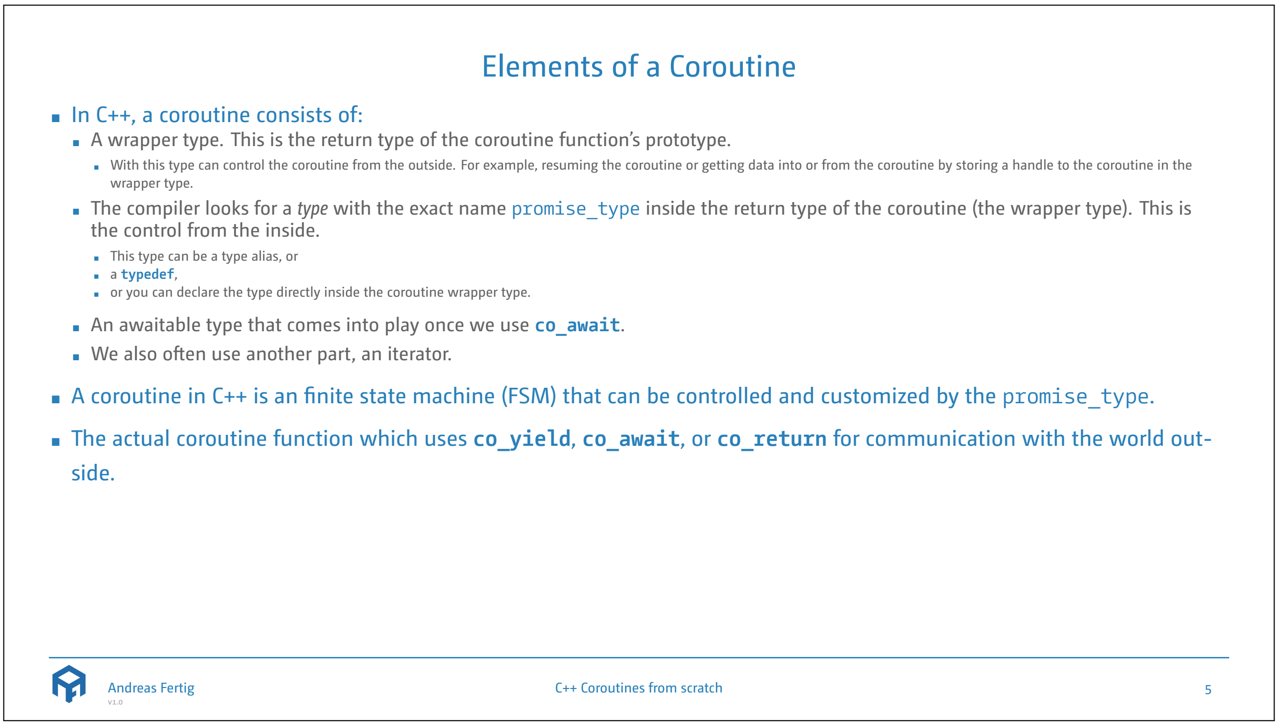
- The term coroutine has been well-established in computer science since it was first coined in 1958 by Melvin Conway [1].
- They come in two different forms:
  - Stack*full*
  - Stack*less* (which is what we have in C++)
- Stackless means that the data of a coroutine, the coroutine frame, is stored on the heap.
- We are talking about cooperative multitasking when using coroutines.
- Coroutines can simplify your code!
  - We can replace some function pointers (callbacks) with coroutines.
  - Parsers are much more readable with coroutines.
  - A lot of state maintenance code is no longer required as the coroutine does the bookkeeping.



Interacting with a coroutine

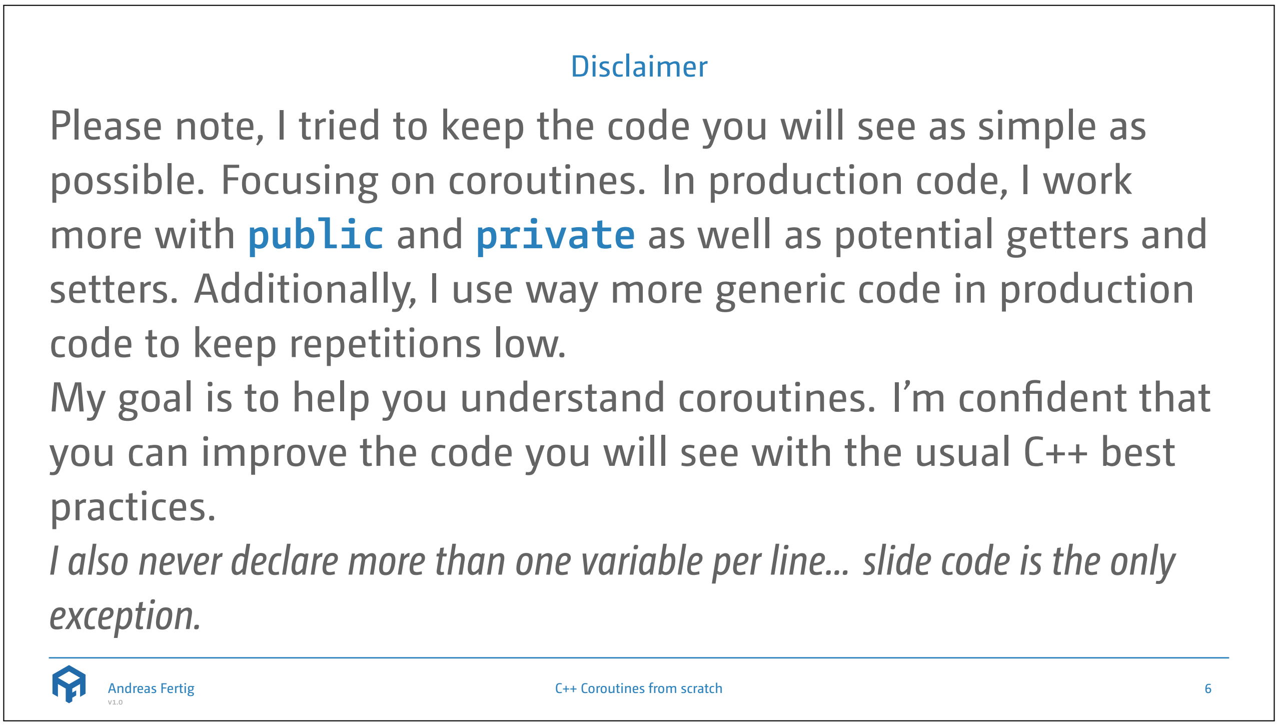
- Coroutines can be paused and resumed.
- `co_yield` or `co_await` pause a coroutine.
- `co_return` ends a coroutine.

Keyword	Action	State
`co_yield`	Output	Suspended
`co_return`	Output	Ended
`co_await`	Input	Suspended



Elements of a Coroutine

- In C++, a coroutine consists of:
  - A wrapper type. This is the return type of the coroutine function's prototype.
    - With this type can control the coroutine from the outside. For example, resuming the coroutine or getting data into or from the coroutine by storing a handle to the coroutine in the wrapper type.
  - The compiler looks for a *type* with the exact name `promise_type` inside the return type of the coroutine (the wrapper type). This is the control from the inside.
    - This type can be a type alias, or
    - a `typedef`,
    - or you can declare the type directly inside the coroutine wrapper type.
  - An awaitable type that comes into play once we use `co_await`.
  - We also often use another part, an iterator.
- A coroutine in C++ is an finite state machine (FSM) that can be controlled and customized by the `promise_type`.
- The actual coroutine function which uses `co_yield`, `co_await`, or `co_return` for communication with the world outside.

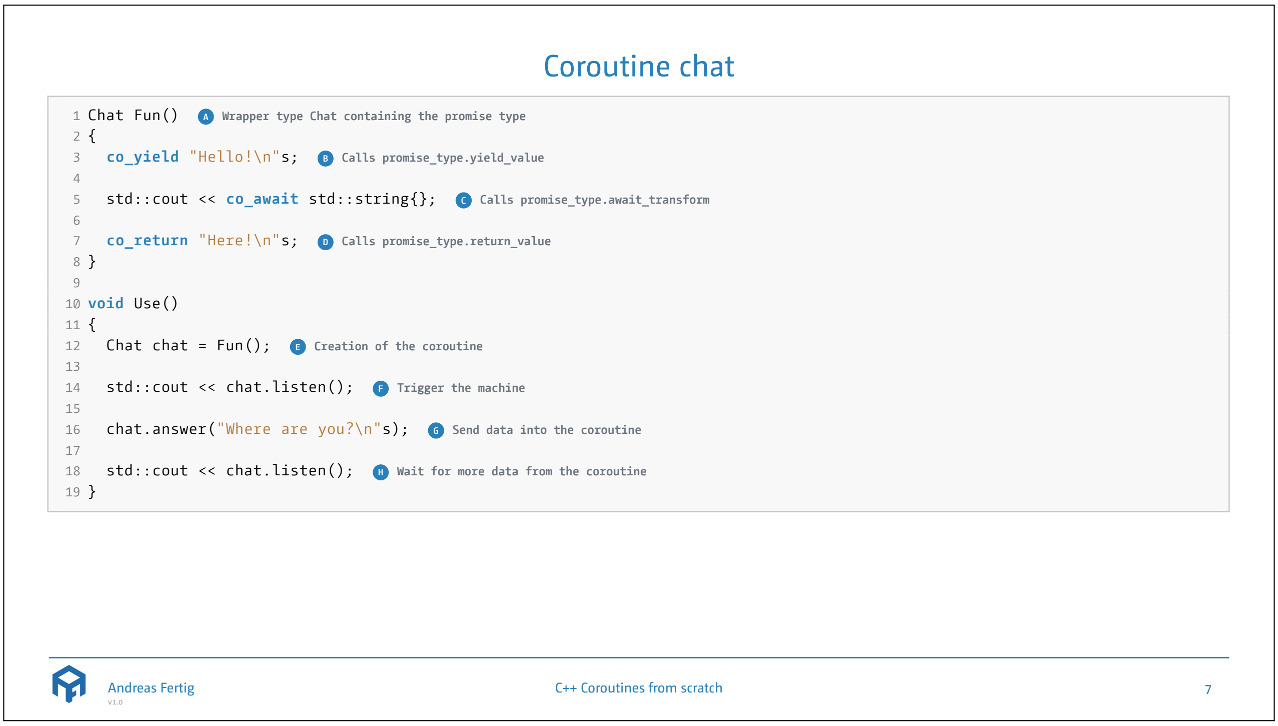


Disclaimer

Please note, I tried to keep the code you will see as simple as possible. Focusing on coroutines. In production code, I work more with **`public`** and **`private`** as well as potential getters and setters. Additionally, I use way more generic code in production code to keep repetitions low.

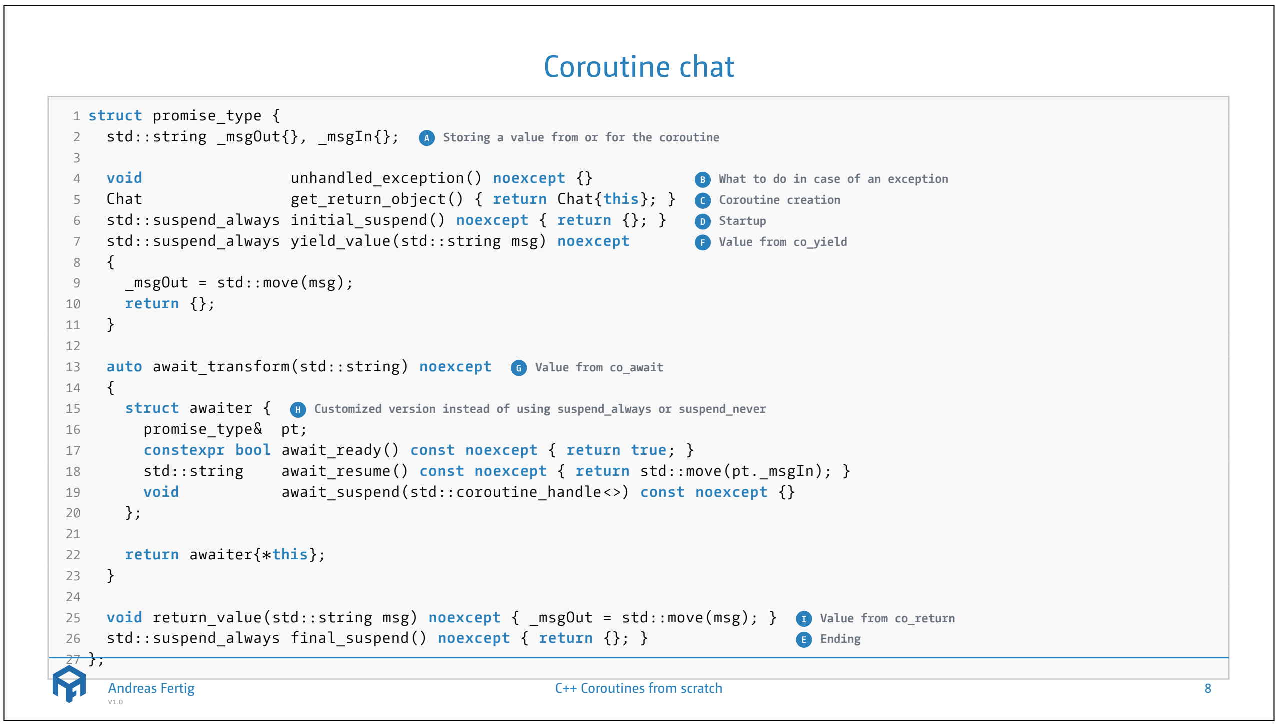
My goal is to help you understand coroutines. I'm confident that you can improve the code you will see with the usual C++ best practices.

I also never declare more than one variable per line... slide code is the only exception.



Coroutine chat

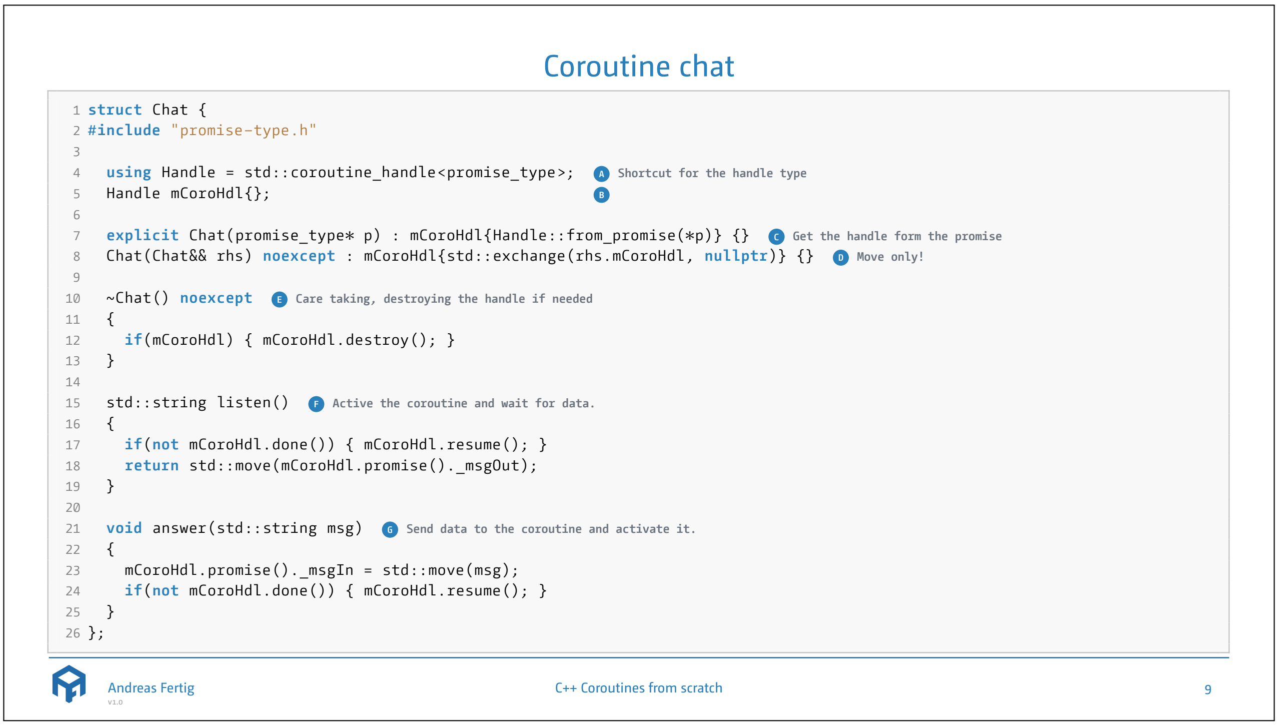
```
Chat Fun()  A Wrapper type Chat containing the promise type
{
  co_yield "Hello!\n"s;  B Calls promise_type.yield_value

  std::cout << co_await std::string{};  C Calls promise_type.await_transform

  co_return "Here!\n"s;  D Calls promise_type.return_value
}

void Use()
{
  Chat chat = Fun();  E Creation of the coroutine

  std::cout << chat.listen();  F Trigger the machine

  chat.answer("Where are you?\n"s);  G Send data into the coroutine

  std::cout << chat.listen();  H Wait for more data from the coroutine
}
```



Coroutine chat

```
struct promise_type {
  std::string _msgOut{}, _msgIn{};  A Storing a value from or for the coroutine

  void                 unhandled_exception() noexcept {}                    B What to do in case of an exception
  Chat                 get_return_object() { return Chat{this}; }           C Coroutine creation
  std::suspend_always initial_suspend() noexcept { return {}; }             D Startup
  std::suspend_always yield_value(std::string msg) noexcept                 F Value from co_yield
  {
    _msgOut = std::move(msg);
    return {};
  }

  auto await_transform(std::string) noexcept  G Value from co_await
  {
    struct awaiter {  H Customized version instead of using suspend_always or suspend_never
      promise_type&  pt;
      constexpr bool await_ready() const noexcept { return true; }
      std::string    await_resume() const noexcept { return std::move(pt._msgIn); }
      void           await_suspend(std::coroutine_handle<>) const noexcept {}
    };

    return awaiter{*this};
  }

  void return_value(std::string msg) noexcept { _msgOut = std::move(msg); }  I Value from co_return
  std::suspend_always final_suspend() noexcept { return {}; }                E Ending
};
```

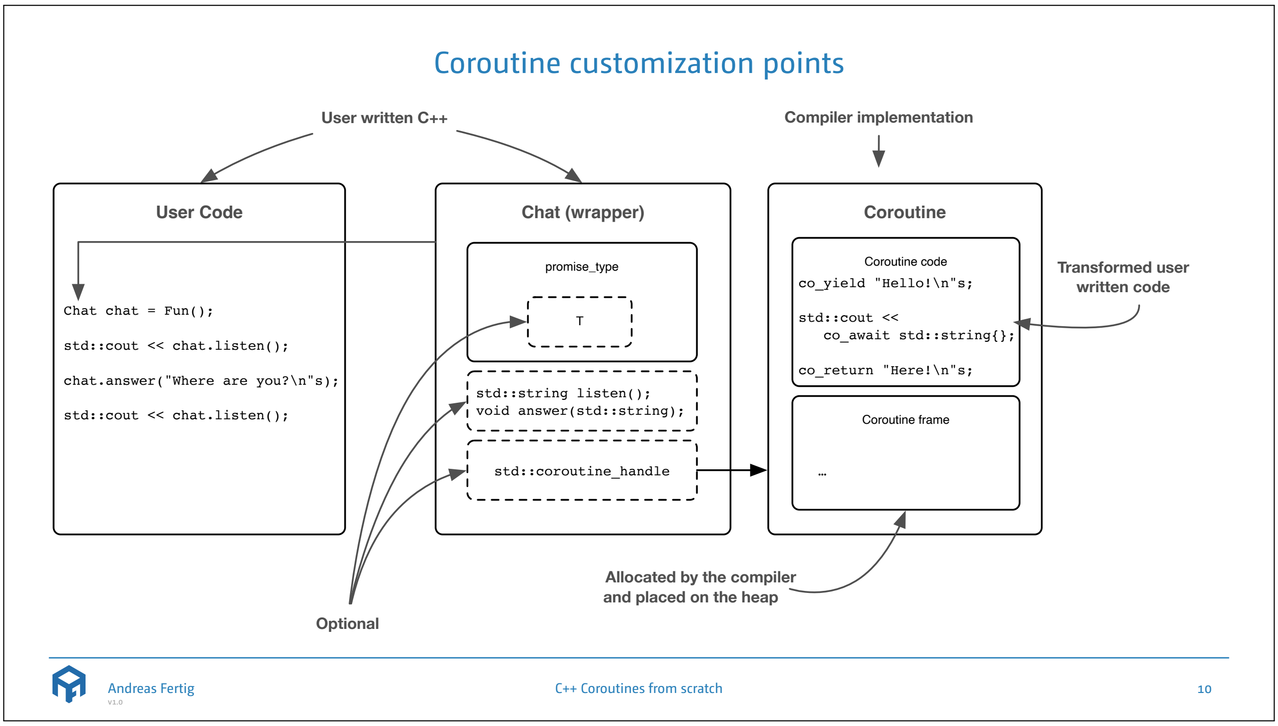
Andreas Fertig v1.0 C++ Coroutines from scratch 8



Coroutine chat

```
struct Chat {
#include "promise-type.h"

  using Handle = std::coroutine_handle<promise_type>;  A Shortcut for the handle type
  Handle mCoroHdl{};                                   B

  explicit Chat(promise_type* p) : mCoroHdl{Handle::from_promise(*p)} {}  C Get the handle form the promise
  Chat(Chat&& rhs) noexcept : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)} {}  D Move only!

  ~Chat() noexcept  E Care taking, destroying the handle if needed
  {
    if(mCoroHdl) { mCoroHdl.destroy(); }
  }

  std::string listen()  F Active the coroutine and wait for data.
  {
    if(not mCoroHdl.done()) { mCoroHdl.resume(); }
    return std::move(mCoroHdl.promise()._msgOut);
  }

  void answer(std::string msg)  G Send data to the coroutine and activate it.
  {
    mCoroHdl.promise()._msgIn = std::move(msg);
    if(not mCoroHdl.done()) { mCoroHdl.resume(); }
  }
};
```

Andreas Fertig v1.0 C++ Coroutines from scratch 9



Coroutine customization points

User written C++

Compiler implementation

User Code

```
Chat chat = Fun();

std::cout << chat.listen();

chat.answer("Where are you?\n"s);

std::cout << chat.listen();
```

Chat (wrapper)

promise_type

T

```
std::string listen();
void answer(std::string);
```

```
std::coroutine_handle
```

Coroutine

Coroutine code

```
co_yield "Hello!\n"s;

std::cout <<
   co_await std::string{};

co_return "Here!\n"s;
```

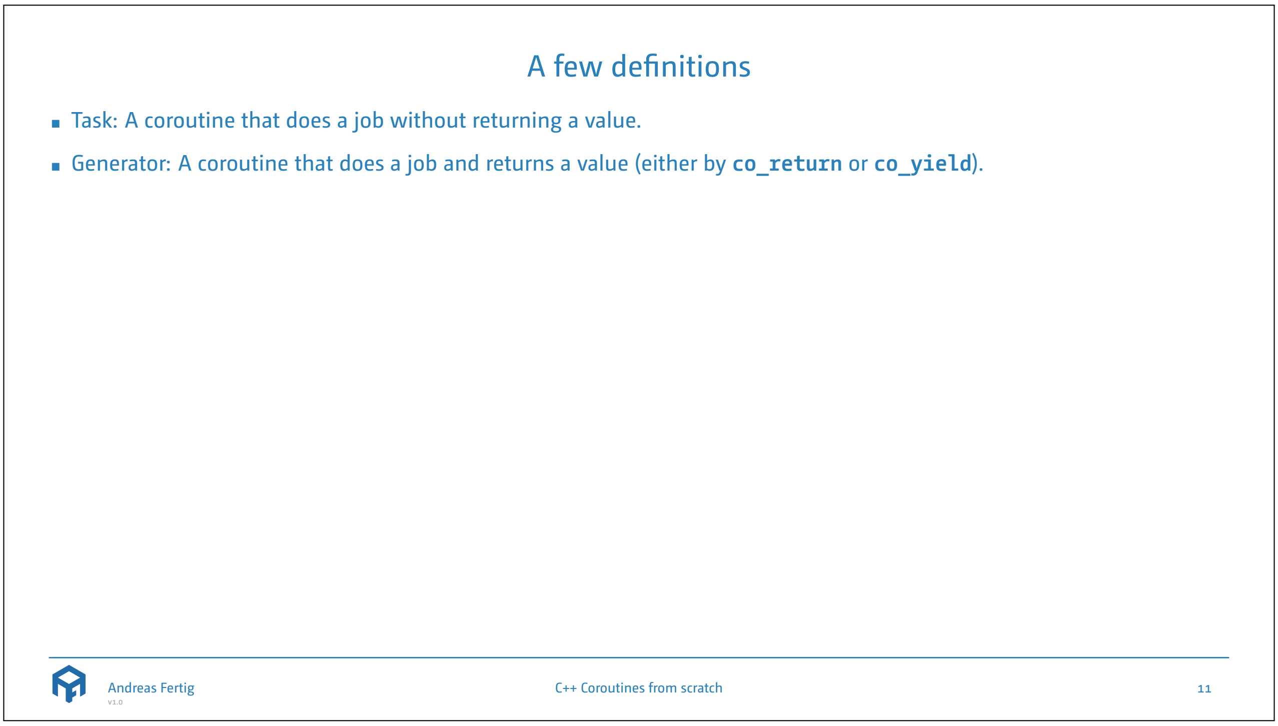
Transformed user written code

Coroutine frame

…

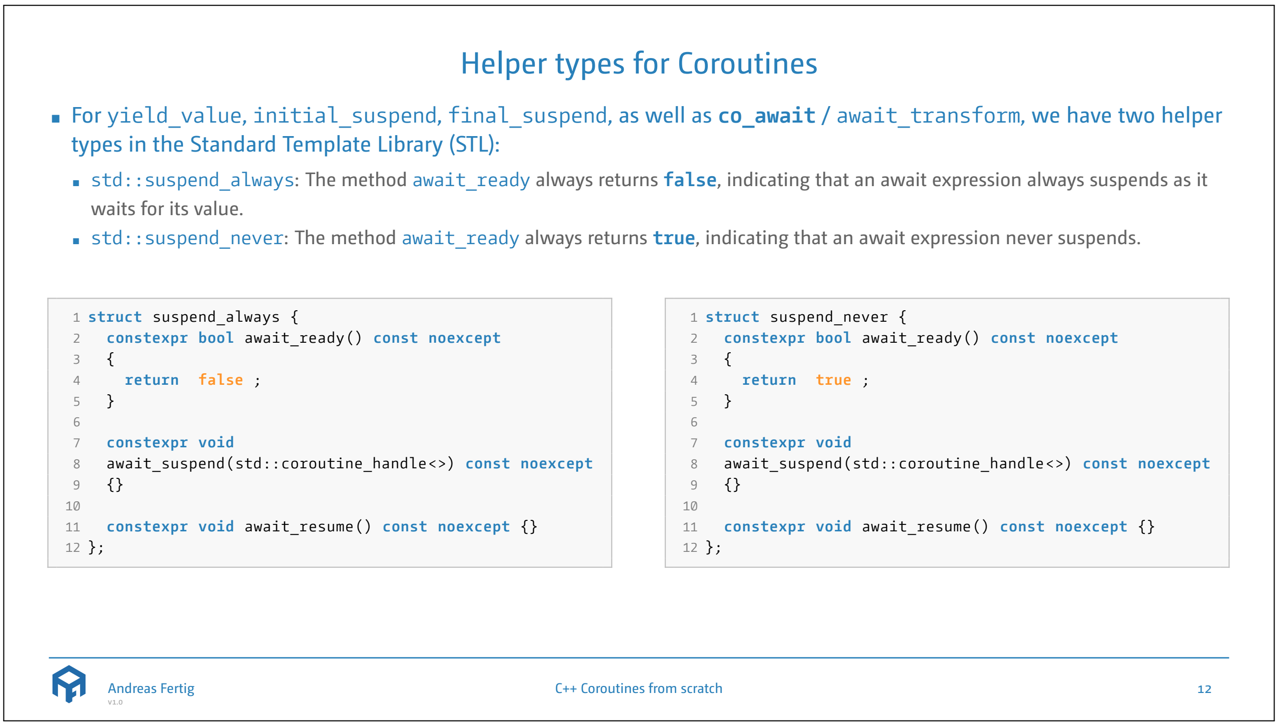
Allocated by the compiler and placed on the heap

Optional



A few definitions

- Task: A coroutine that does a job without returning a value.
- Generator: A coroutine that does a job and returns a value (either by `co_return` or `co_yield`).

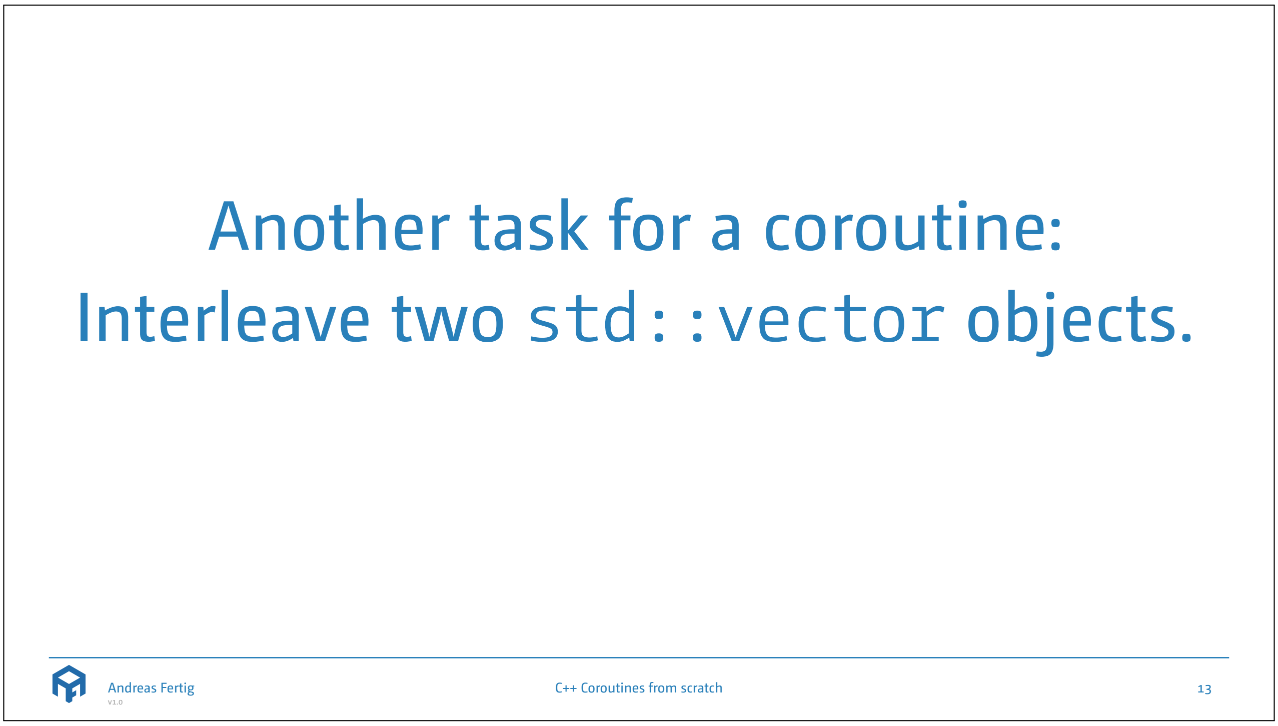


Helper types for Coroutines

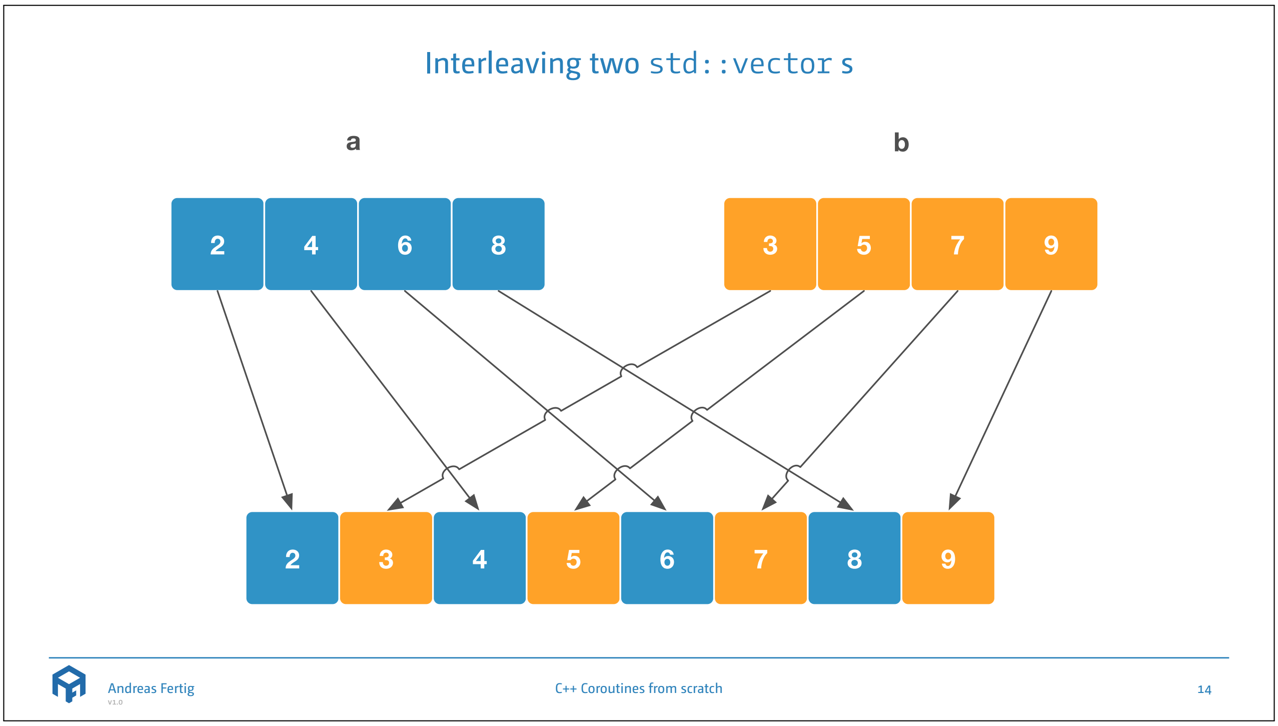
- For `yield_value`, `initial_suspend`, `final_suspend`, as well as **`co_await`** / `await_transform`, we have two helper types in the Standard Template Library (STL):
  - `std::suspend_always`: The method `await_ready` always returns **`false`**, indicating that an await expression always suspends as it waits for its value.
  - `std::suspend_never`: The method `await_ready` always returns **`true`**, indicating that an await expression never suspends.

```cpp
struct suspend_always {
  constexpr bool await_ready() const noexcept
  {
    return  false ;
  }

  constexpr void
  await_suspend(std::coroutine_handle<>) const noexcept
  {}

  constexpr void await_resume() const noexcept {}
};
```

```cpp
struct suspend_never {
  constexpr bool await_ready() const noexcept
  {
    return  true ;
  }

  constexpr void
  await_suspend(std::coroutine_handle<>) const noexcept
  {}

  constexpr void await_resume() const noexcept {}
};
```



Another task for a coroutine: Interleave two `std::vector` objects.



Interleaving two `std::vector` s

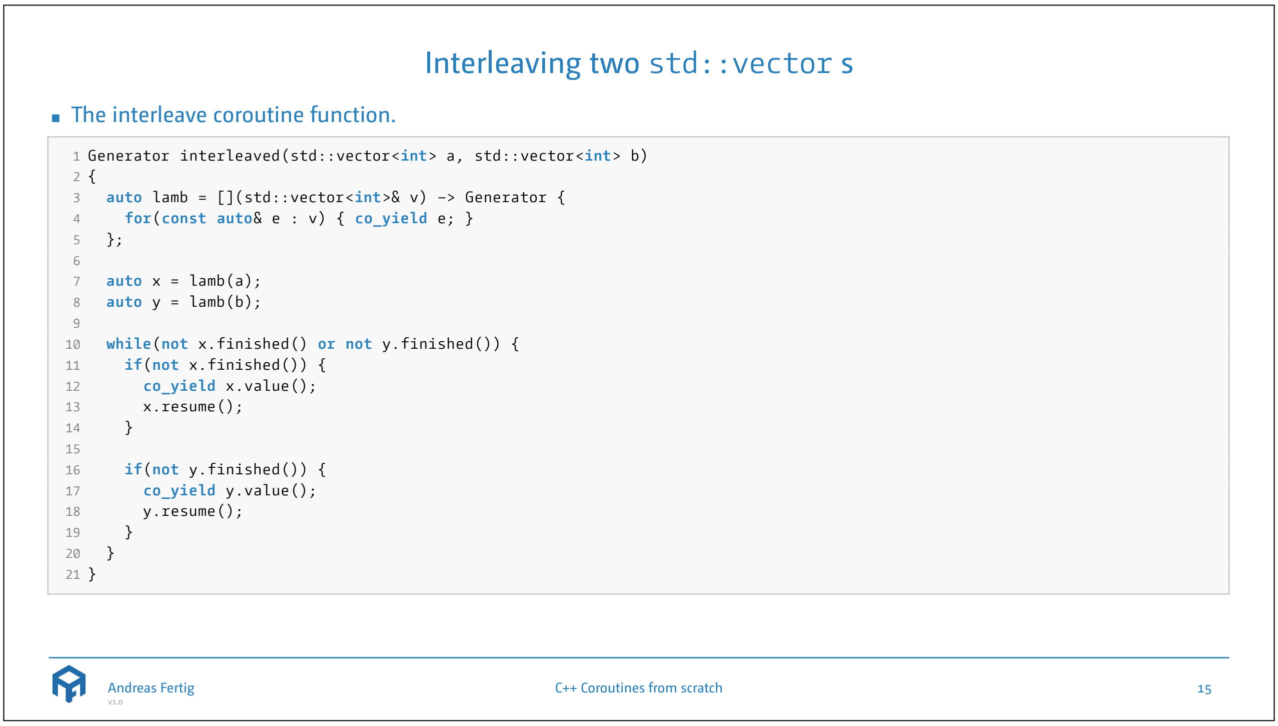
a

2 4 6 8

b

3 5 7 9

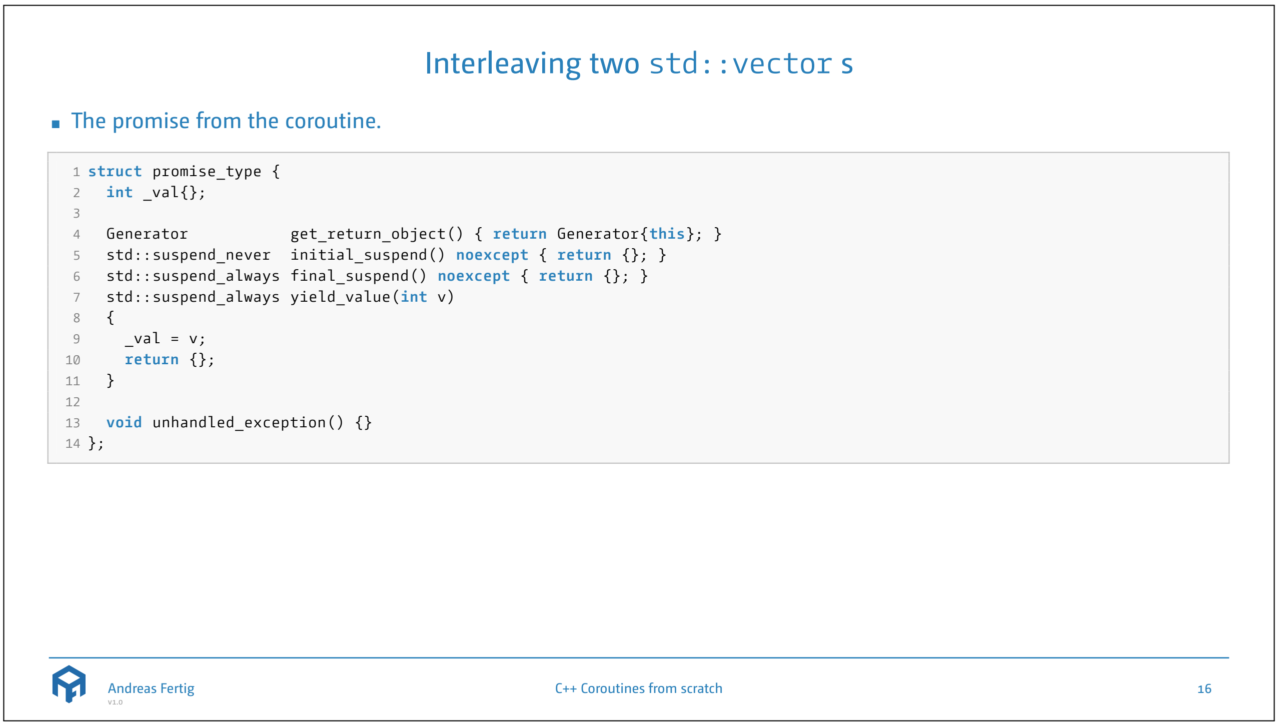
2 3 4 5 6 7 8 9



Interleaving two `std::vector` s

- The interleave coroutine function.

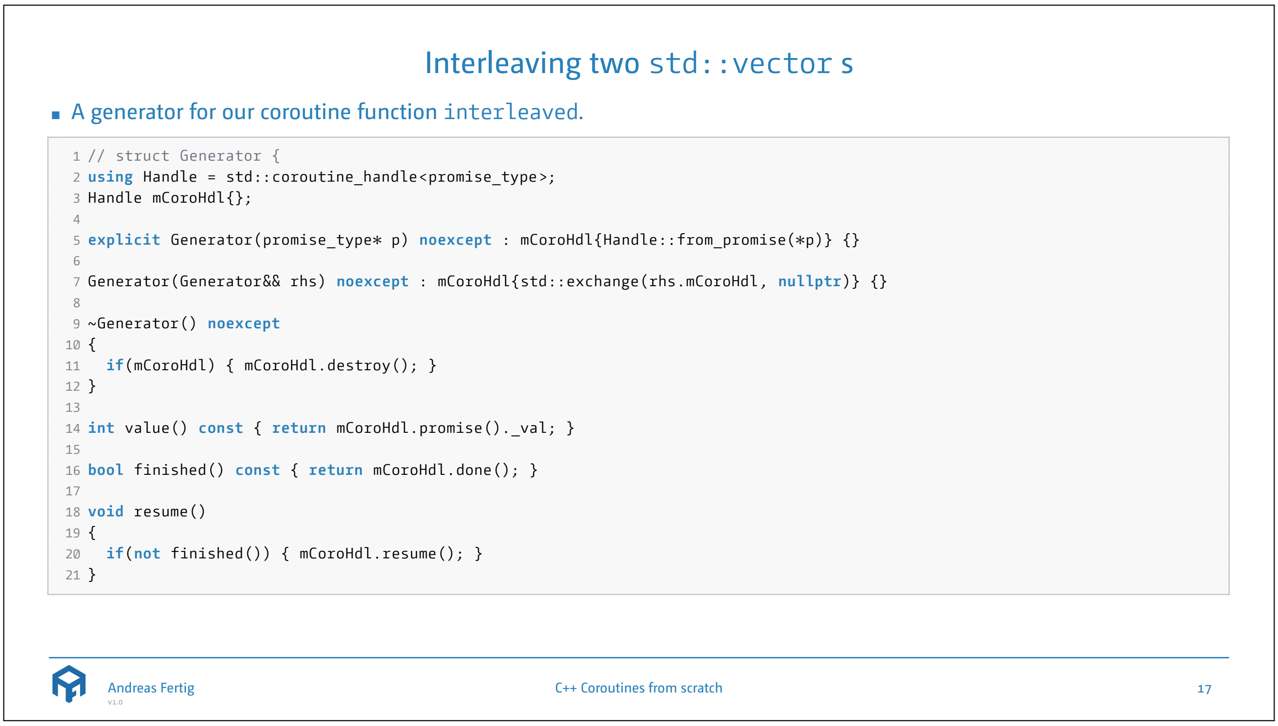
```cpp
Generator interleaved(std::vector<int> a, std::vector<int> b)
{
  auto lamb = [](std::vector<int>& v) -> Generator {
    for(const auto& e : v) { co_yield e; }
  };

  auto x = lamb(a);
  auto y = lamb(b);

  while(not x.finished() or not y.finished()) {
    if(not x.finished()) {
      co_yield x.value();
      x.resume();
    }

    if(not y.finished()) {
      co_yield y.value();
      y.resume();
    }
  }
}
```



Interleaving two std::vector s

- The promise from the coroutine.

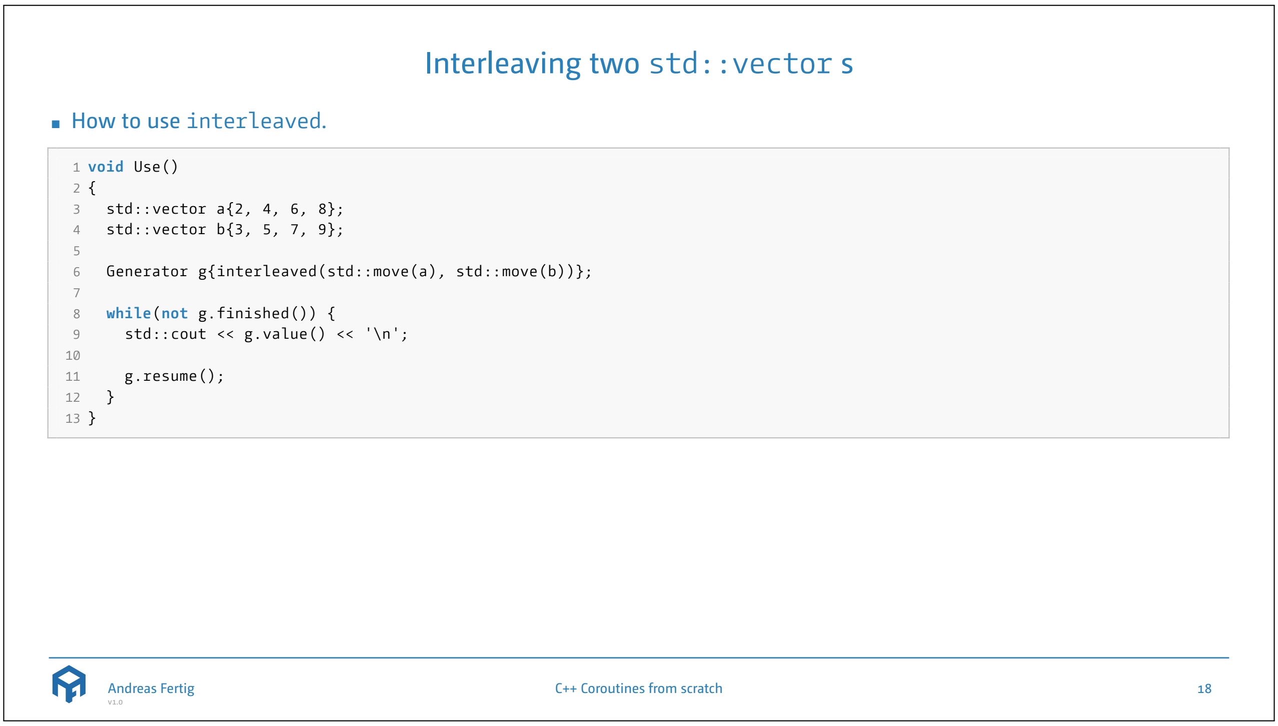
```cpp
struct promise_type {
  int _val{};

  Generator           get_return_object() { return Generator{this}; }
  std::suspend_never  initial_suspend() noexcept { return {}; }
  std::suspend_always final_suspend() noexcept { return {}; }
  std::suspend_always yield_value(int v)
  {
    _val = v;
    return {};
  }

  void unhandled_exception() {}
};
```



Interleaving two std::vector s

- A generator for our coroutine function interleaved.

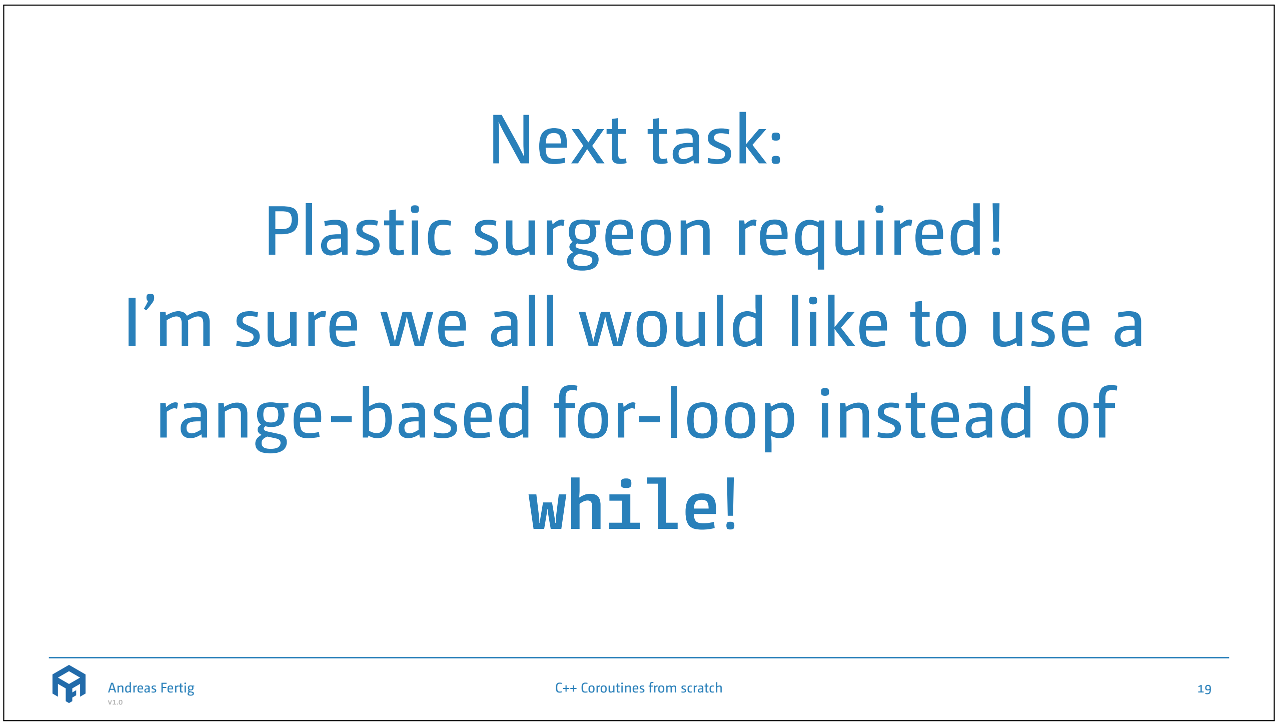
```cpp
// struct Generator {
using Handle = std::coroutine_handle<promise_type>;
Handle mCoroHdl{};

explicit Generator(promise_type* p) noexcept : mCoroHdl{Handle::from_promise(*p)} {}

Generator(Generator&& rhs) noexcept : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)} {}

~Generator() noexcept
{
  if(mCoroHdl) { mCoroHdl.destroy(); }
}

int value() const { return mCoroHdl.promise()._val; }

bool finished() const { return mCoroHdl.done(); }

void resume()
{
  if(not finished()) { mCoroHdl.resume(); }
}
```



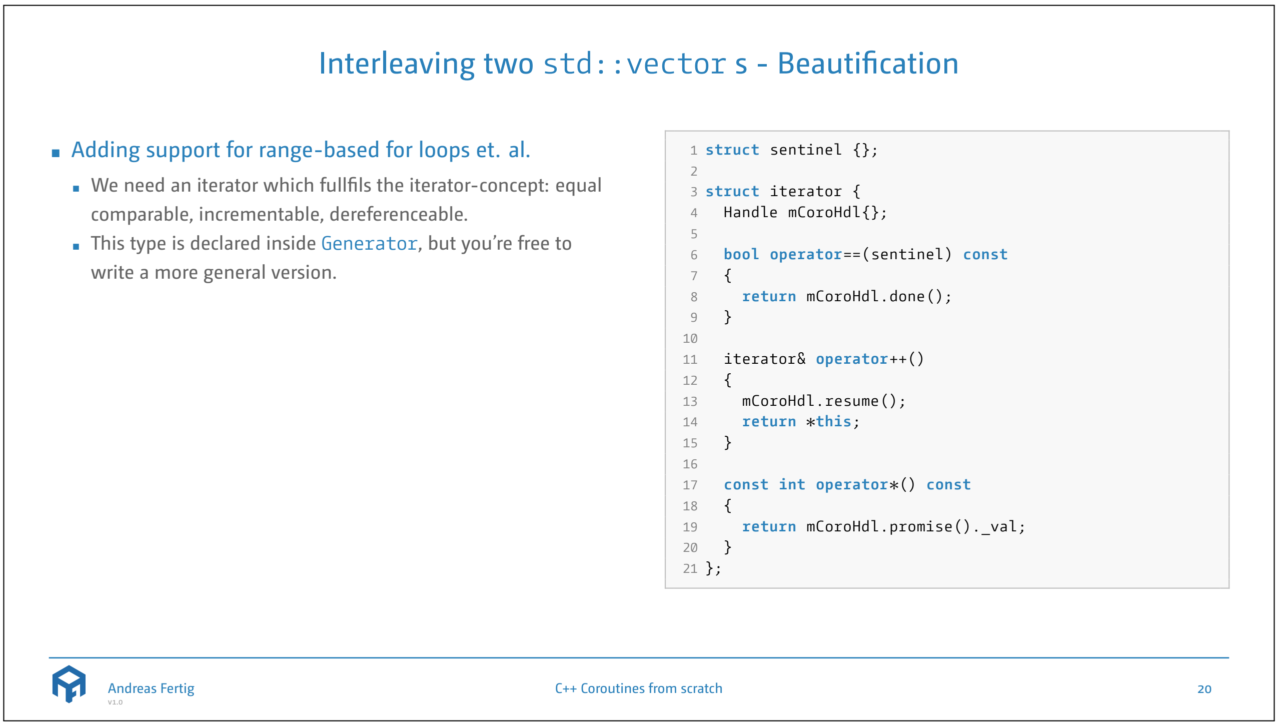
Interleaving two `std::vector` s

- How to use `interleaved`.

```cpp
void Use()
{
  std::vector a{2, 4, 6, 8};
  std::vector b{3, 5, 7, 9};

  Generator g{interleaved(std::move(a), std::move(b))};

  while(not g.finished()) {
    std::cout << g.value() << '\n';

    g.resume();
  }
}
```



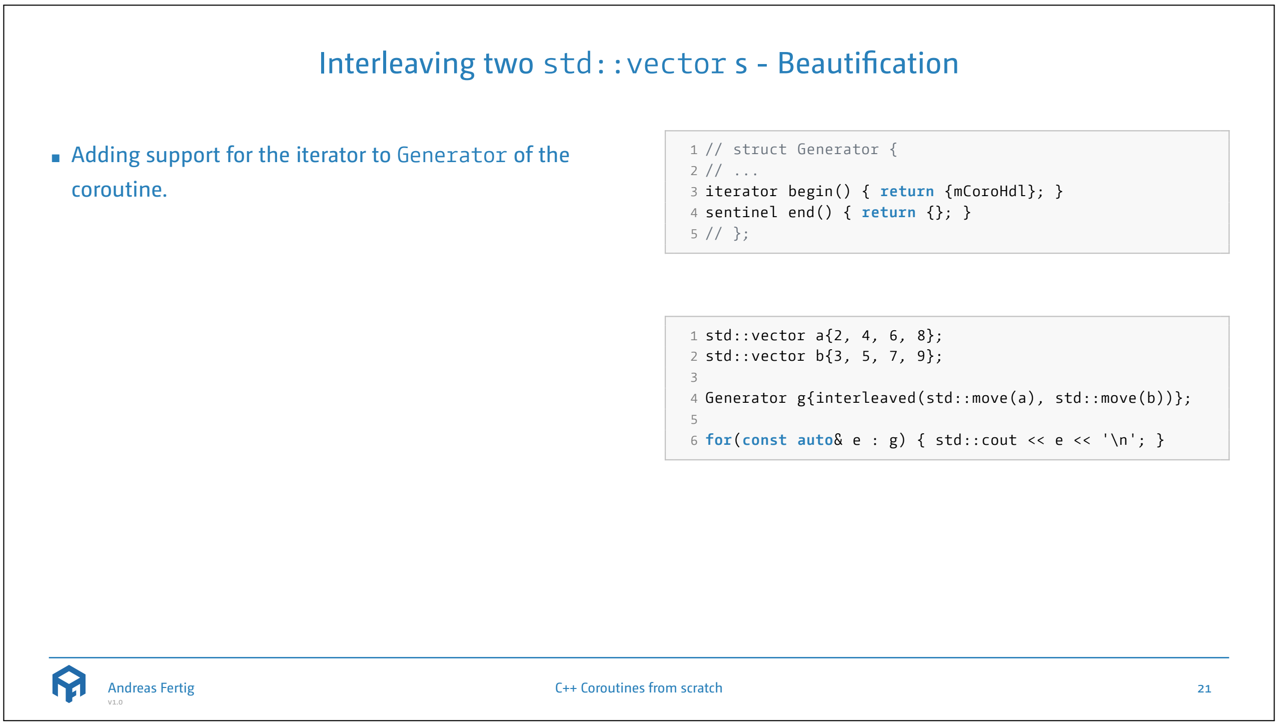
Next task:
Plastic surgeon required!
I’m sure we all would like to use a range-based for-loop instead of **`while`**!



Interleaving two std::vector s - Beautification

- Adding support for range-based for loops et. al.
  - We need an iterator which fullfils the iterator-concept: equal comparable, incrementable, dereferenceable.
  - This type is declared inside Generator, but you're free to write a more general version.

```cpp
struct sentinel {};

struct iterator {
  Handle mCoroHdl{};

  bool operator==(sentinel) const
  {
    return mCoroHdl.done();
  }

  iterator& operator++()
  {
    mCoroHdl.resume();
    return *this;
  }

  const int operator*() const
  {
    return mCoroHdl.promise()._val;
  }
};
```

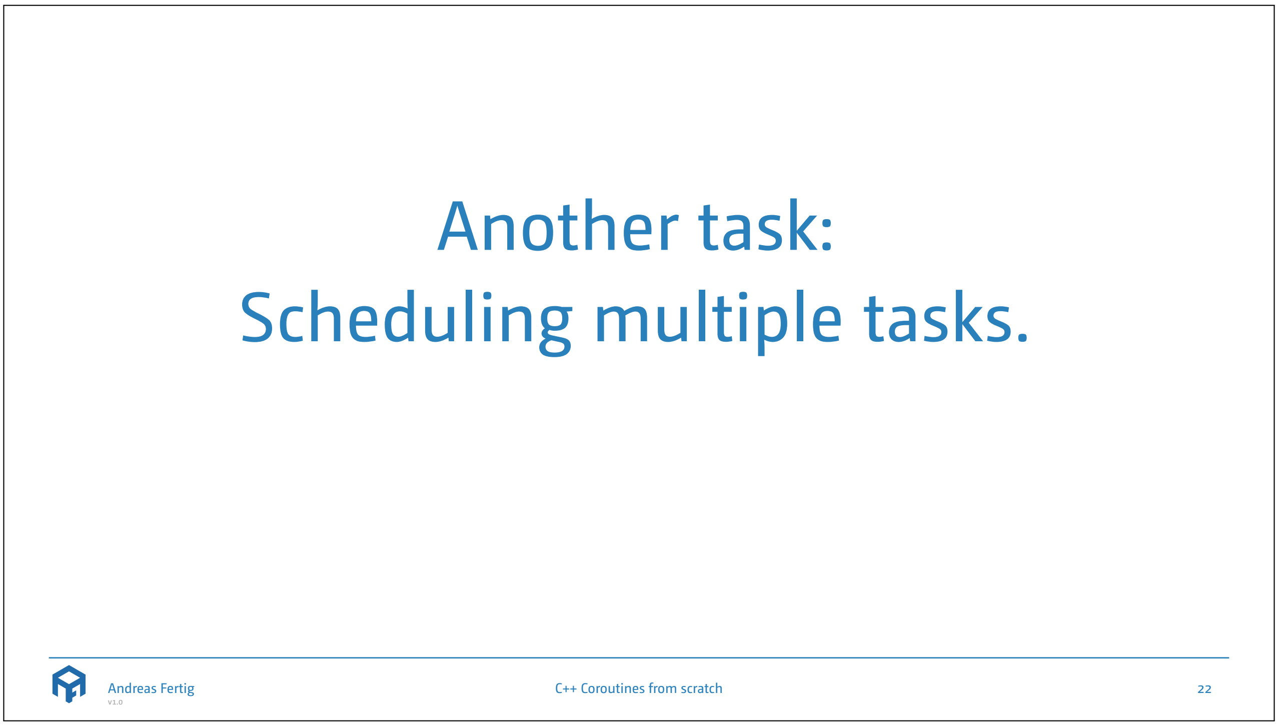


Interleaving two std::vector s - Beautification

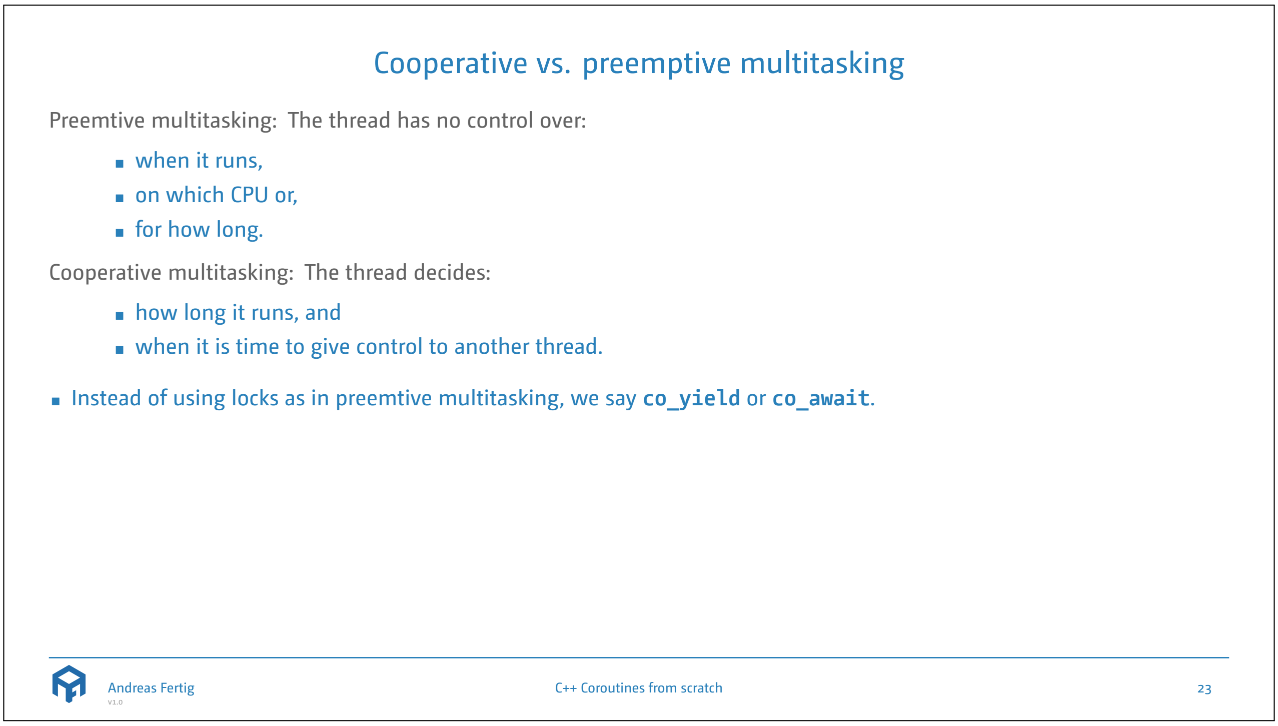
- Adding support for the iterator to Generator of the coroutine.

```cpp
// struct Generator {
// ...
iterator begin() { return {mCoroHdl}; }
sentinel end() { return {}; }
// };
```

```cpp
std::vector a{2, 4, 6, 8};
std::vector b{3, 5, 7, 9};

Generator g{interleaved(std::move(a), std::move(b))};

for(const auto& e : g) { std::cout << e << '\n'; }
```



Another task:
Scheduling multiple tasks.



Cooperative vs. preemptive multitasking

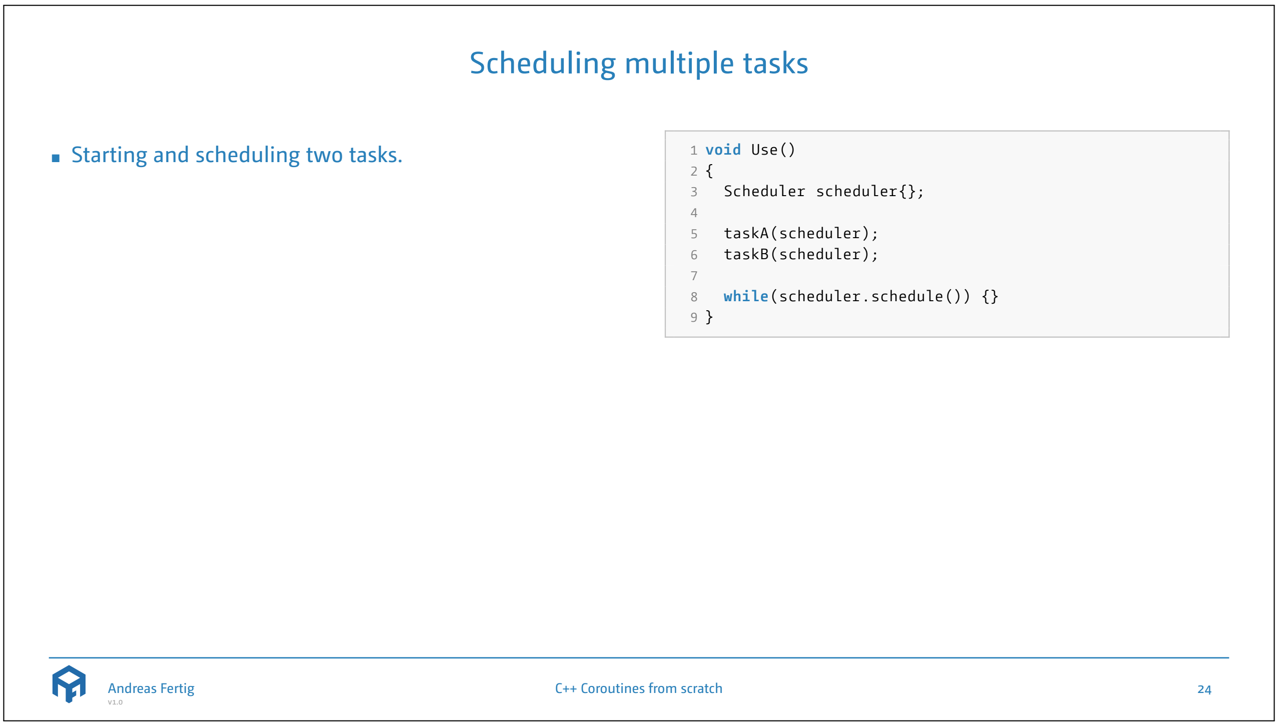
Preemtive multitasking: The thread has no control over:

- when it runs,
- on which CPU or,
- for how long.

Cooperative multitasking: The thread decides:

- how long it runs, and
- when it is time to give control to another thread.

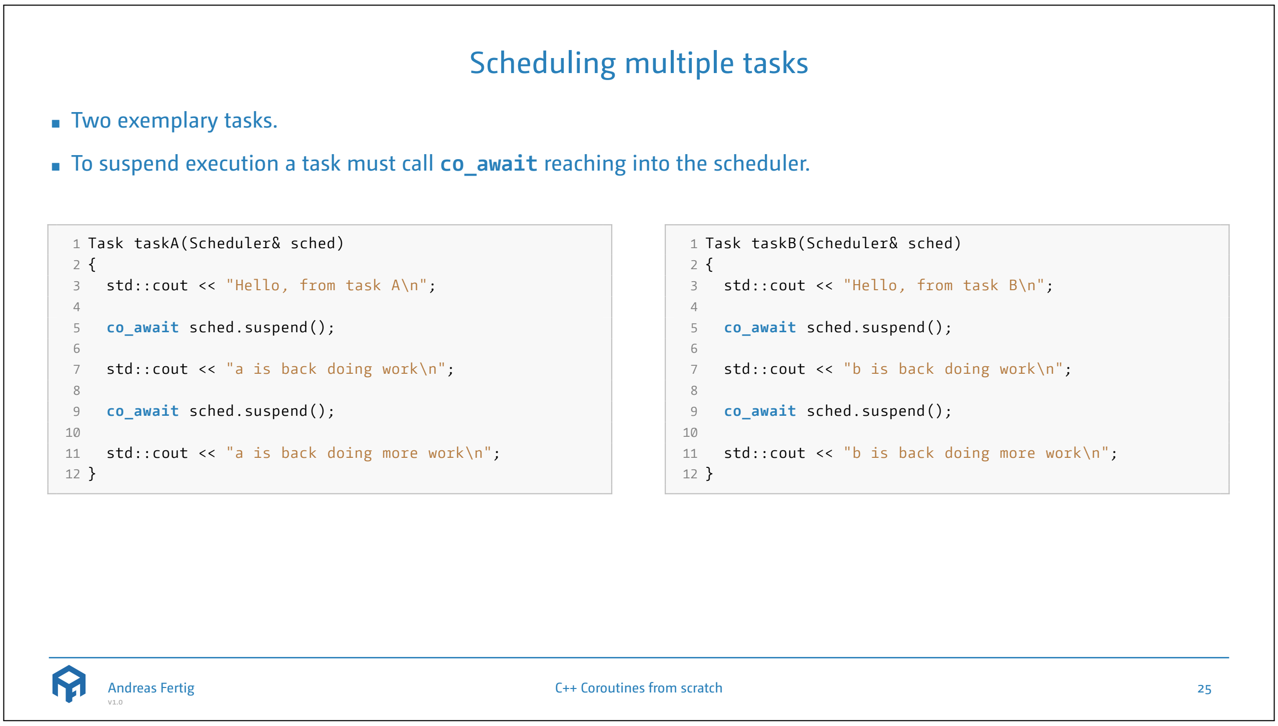
- Instead of using locks as in preemtive multitasking, we say **`co_yield`** or **`co_await`**.



Scheduling multiple tasks

- Starting and scheduling two tasks.

```cpp
void Use()
{
  Scheduler scheduler{};

  taskA(scheduler);
  taskB(scheduler);

  while(scheduler.schedule()) {}
}
```

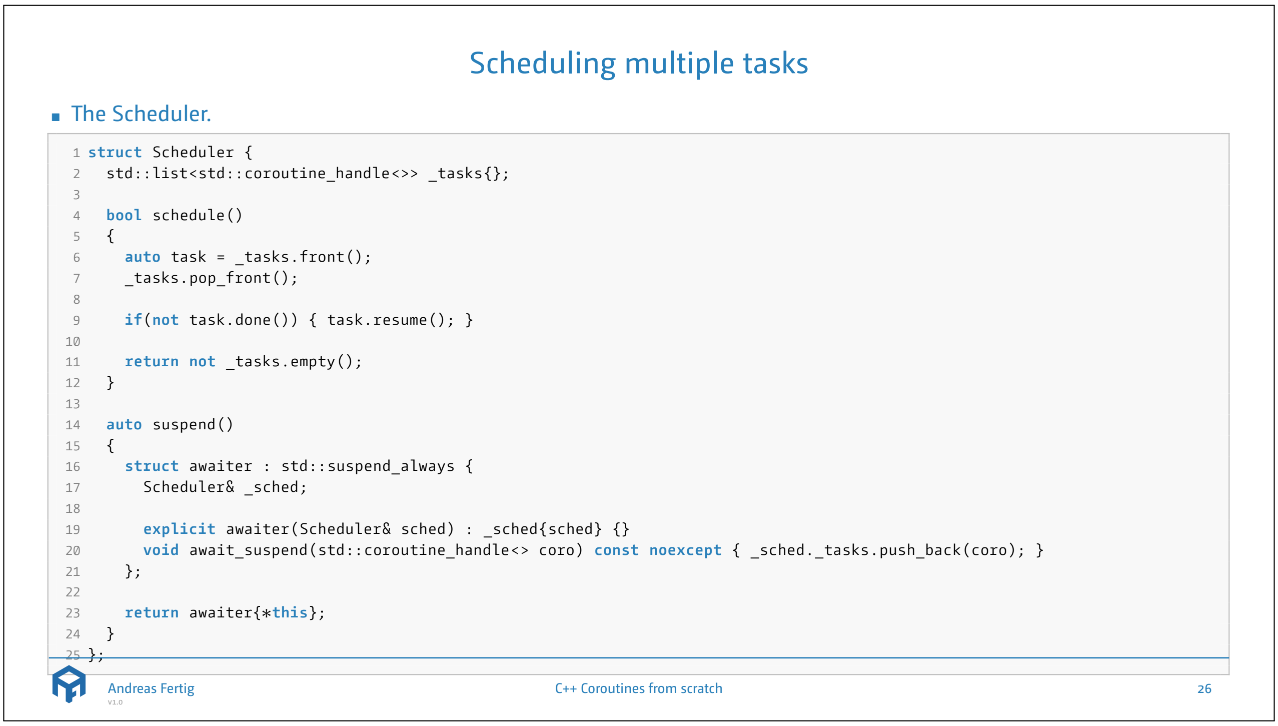


Scheduling multiple tasks

- Two exemplary tasks.
- To suspend execution a task must call `co_await` reaching into the scheduler.

```cpp
Task taskA(Scheduler& sched)
{
  std::cout << "Hello, from task A\n";

  co_await sched.suspend();

  std::cout << "a is back doing work\n";

  co_await sched.suspend();

  std::cout << "a is back doing more work\n";
}
```

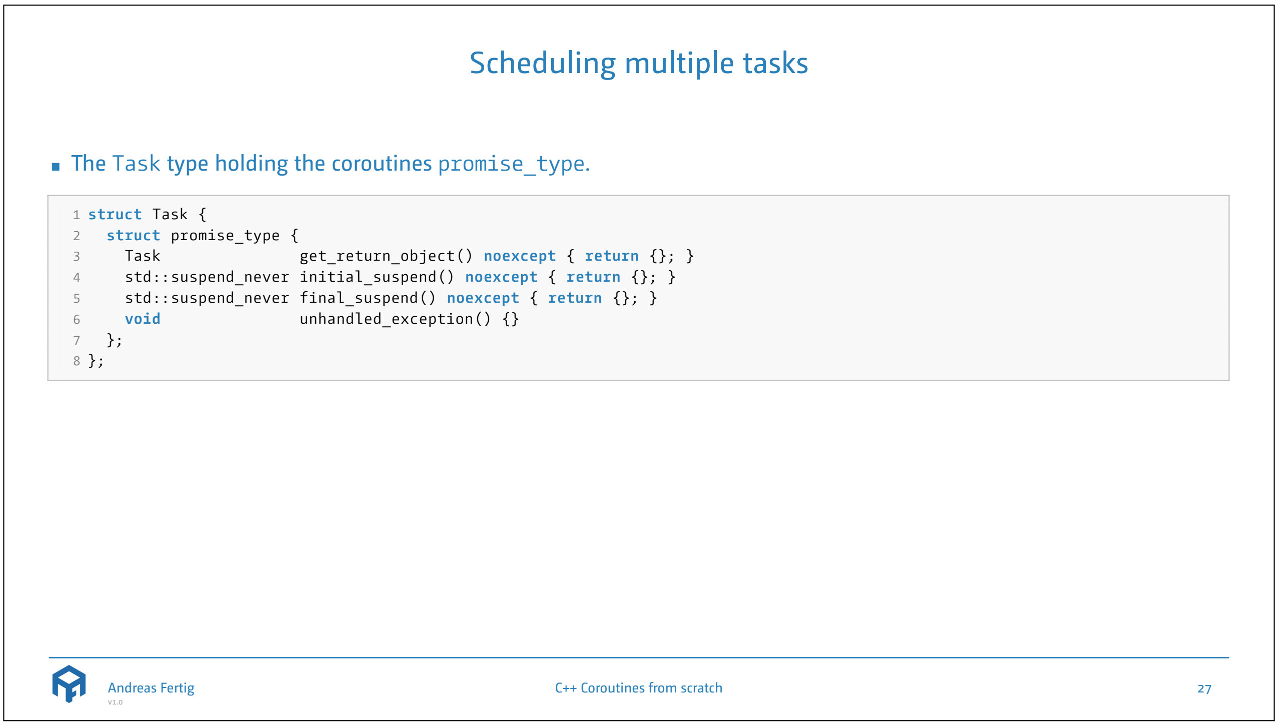
```cpp
Task taskB(Scheduler& sched)
{
  std::cout << "Hello, from task B\n";

  co_await sched.suspend();

  std::cout << "b is back doing work\n";

  co_await sched.suspend();

  std::cout << "b is back doing more work\n";
}
```



Scheduling multiple tasks

- The Scheduler.

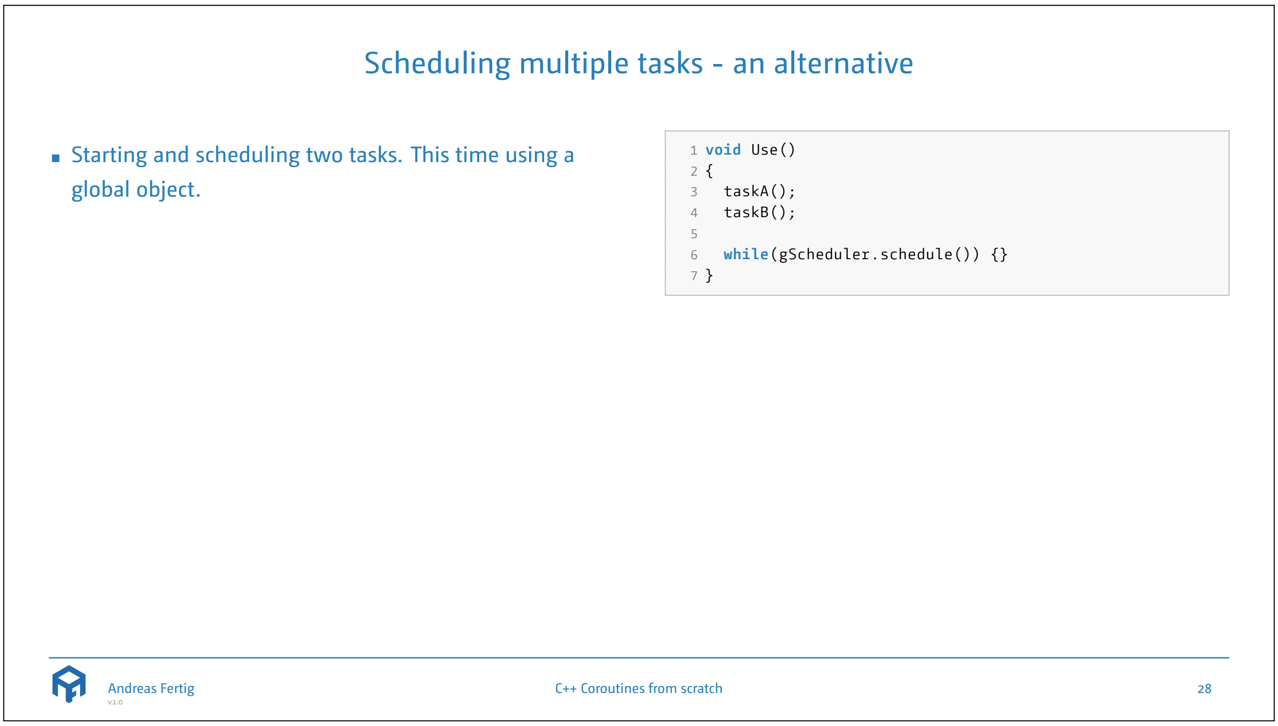
```cpp
struct Scheduler {
  std::list<std::coroutine_handle<>> _tasks{};

  bool schedule()
  {
    auto task = _tasks.front();
    _tasks.pop_front();

    if(not task.done()) { task.resume(); }

    return not _tasks.empty();
  }

  auto suspend()
  {
    struct awaiter : std::suspend_always {
      Scheduler& _sched;

      explicit awaiter(Scheduler& sched) : _sched{sched} {}
      void await_suspend(std::coroutine_handle<> coro) const noexcept { _sched._tasks.push_back(coro); }
    };

    return awaiter{*this};
  }
};
```



Scheduling multiple tasks

- The Task type holding the coroutines `promise_type`.

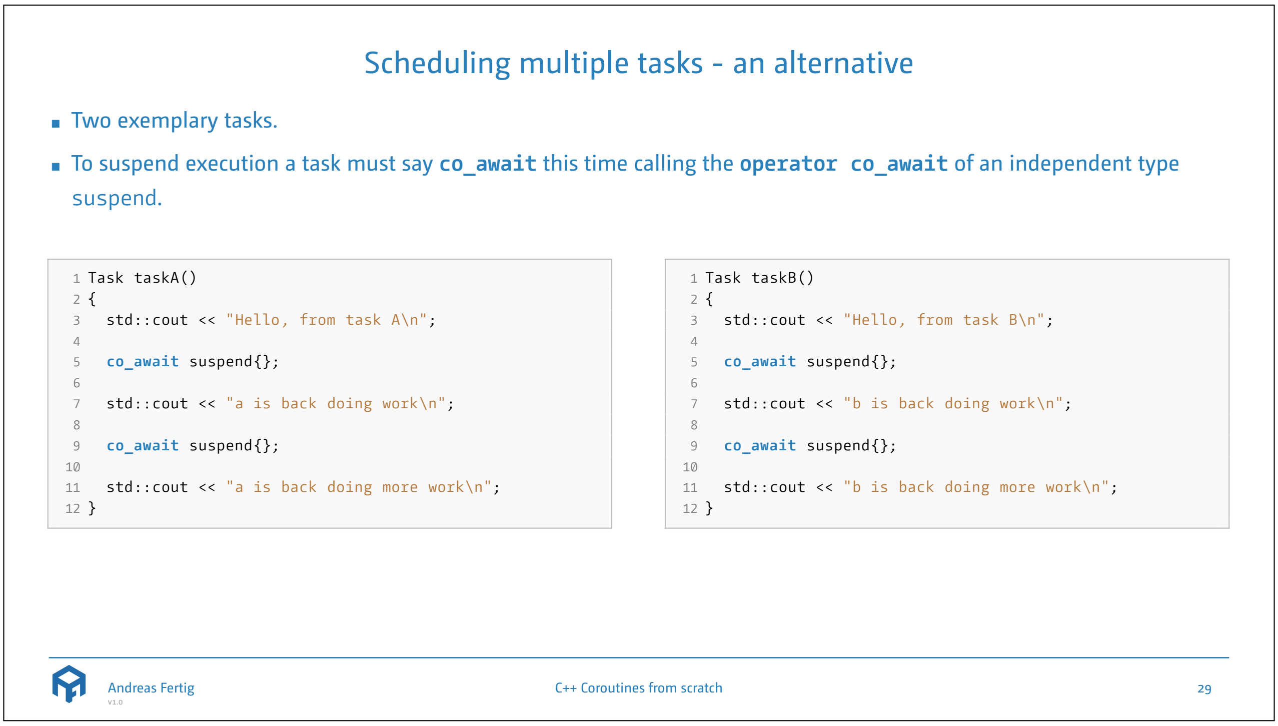
```cpp
struct Task {
  struct promise_type {
    Task                get_return_object() noexcept { return {}; }
    std::suspend_never initial_suspend() noexcept { return {}; }
    std::suspend_never final_suspend() noexcept { return {}; }
    void                unhandled_exception() {}
  };
};
```



Scheduling multiple tasks - an alternative

- Starting and scheduling two tasks. This time using a global object.

```cpp
void Use()
{
  taskA();
  taskB();

  while(gScheduler.schedule()) {}
}
```

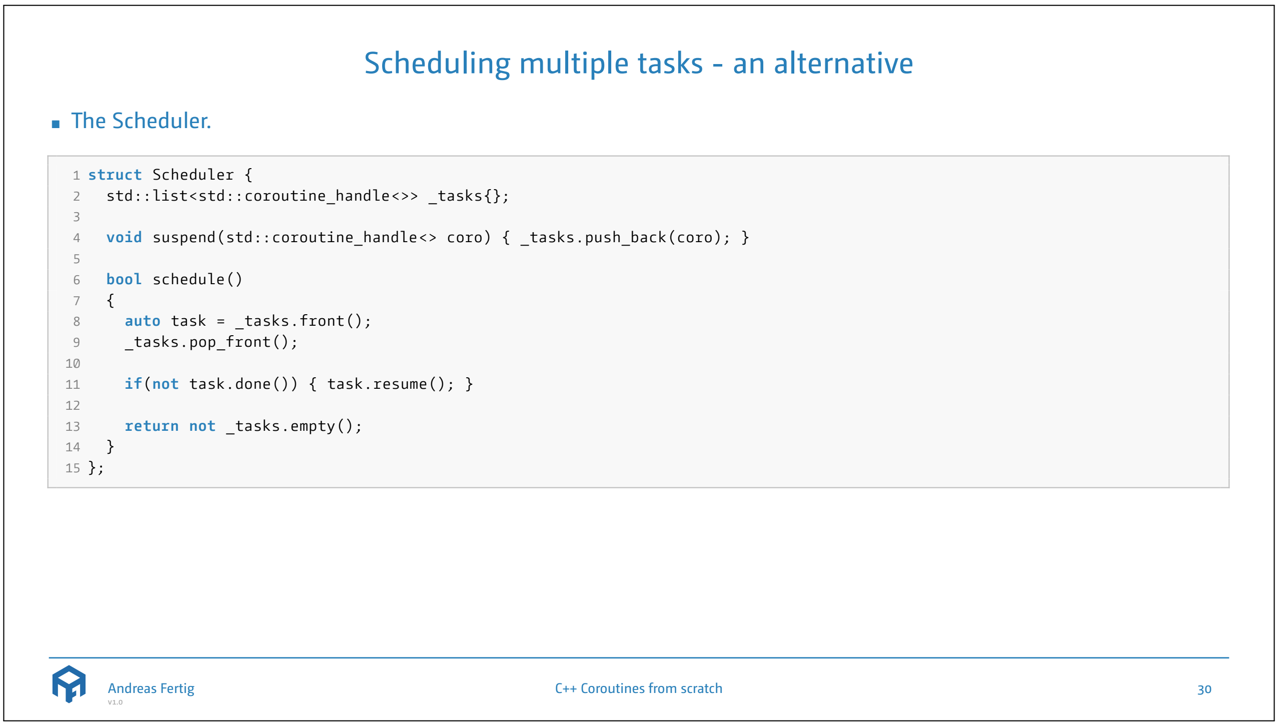


Scheduling multiple tasks - an alternative

- Two exemplary tasks.
- To suspend execution a task must say **`co_await`** this time calling the **`operator co_await`** of an independent type `suspend`.

```cpp
Task taskA()
{
  std::cout << "Hello, from task A\n";

  co_await suspend{};

  std::cout << "a is back doing work\n";

  co_await suspend{};

  std::cout << "a is back doing more work\n";
}
```

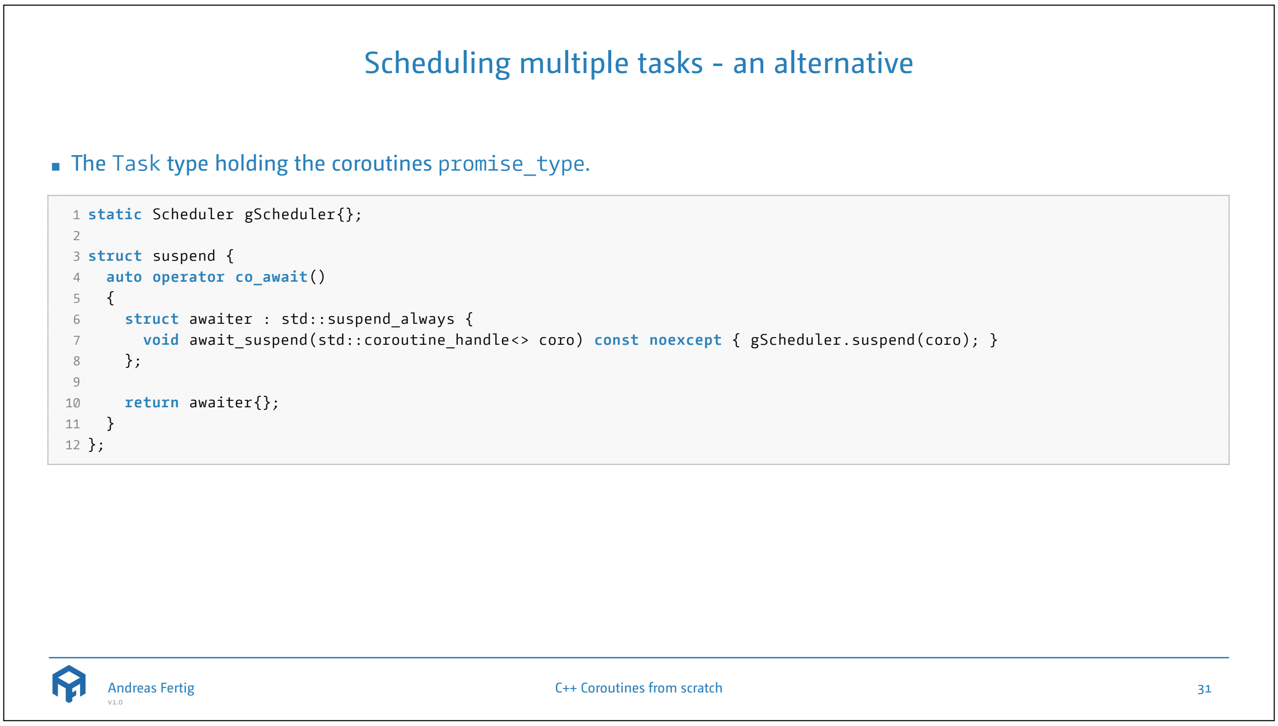
```cpp
Task taskB()
{
  std::cout << "Hello, from task B\n";

  co_await suspend{};

  std::cout << "b is back doing work\n";

  co_await suspend{};

  std::cout << "b is back doing more work\n";
}
```



Scheduling multiple tasks - an alternative

- The Scheduler.

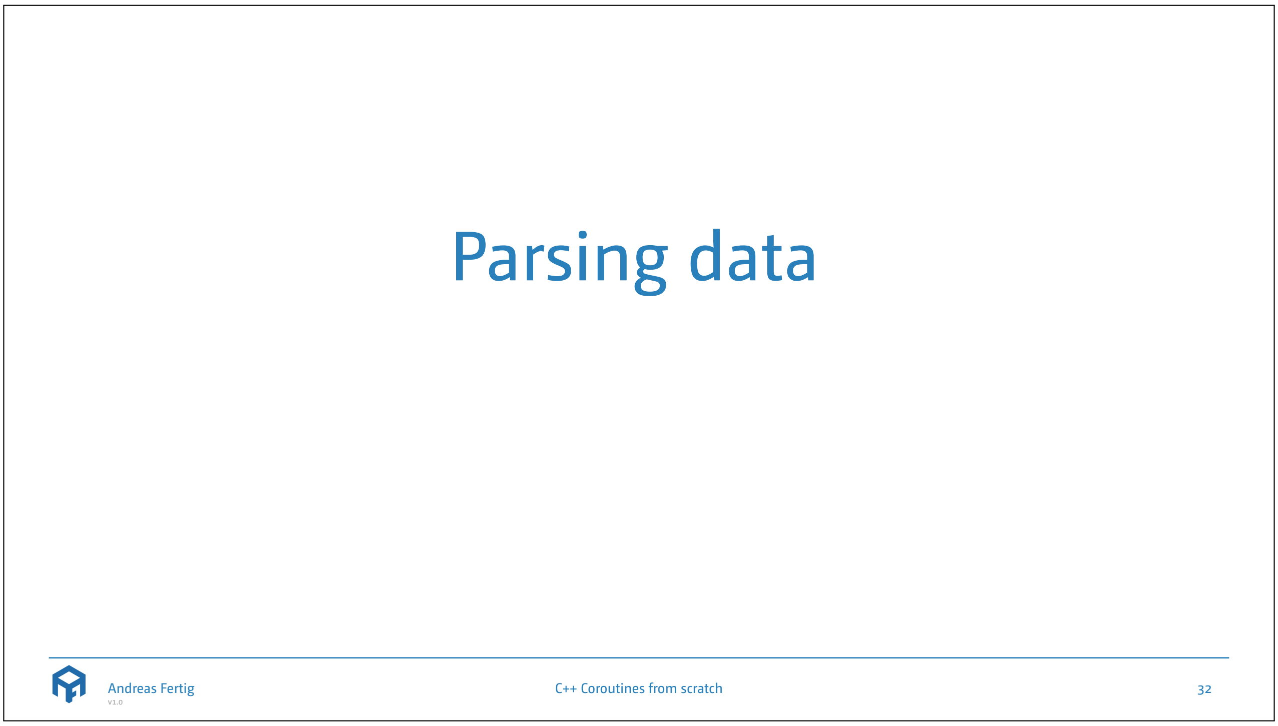
```cpp
struct Scheduler {
  std::list<std::coroutine_handle<>> _tasks{};

  void suspend(std::coroutine_handle<> coro) { _tasks.push_back(coro); }

  bool schedule()
  {
    auto task = _tasks.front();
    _tasks.pop_front();

    if(not task.done()) { task.resume(); }

    return not _tasks.empty();
  }
};
```



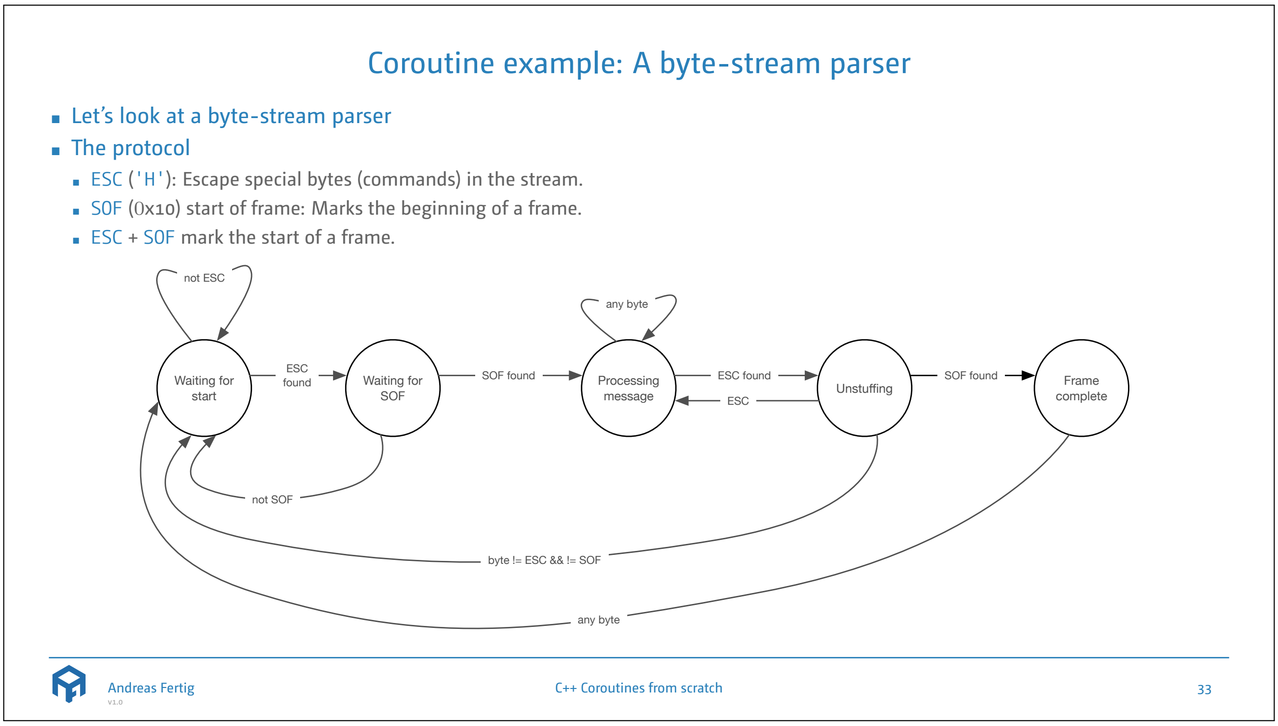
Scheduling multiple tasks - an alternative

- The Task type holding the coroutines promise_type.

```cpp
static Scheduler gScheduler{};

struct suspend {
  auto operator co_await()
  {
    struct awaiter : std::suspend_always {
      void await_suspend(std::coroutine_handle<> coro) const noexcept { gScheduler.suspend(coro); }
    };

    return awaiter{};
  }
};
```

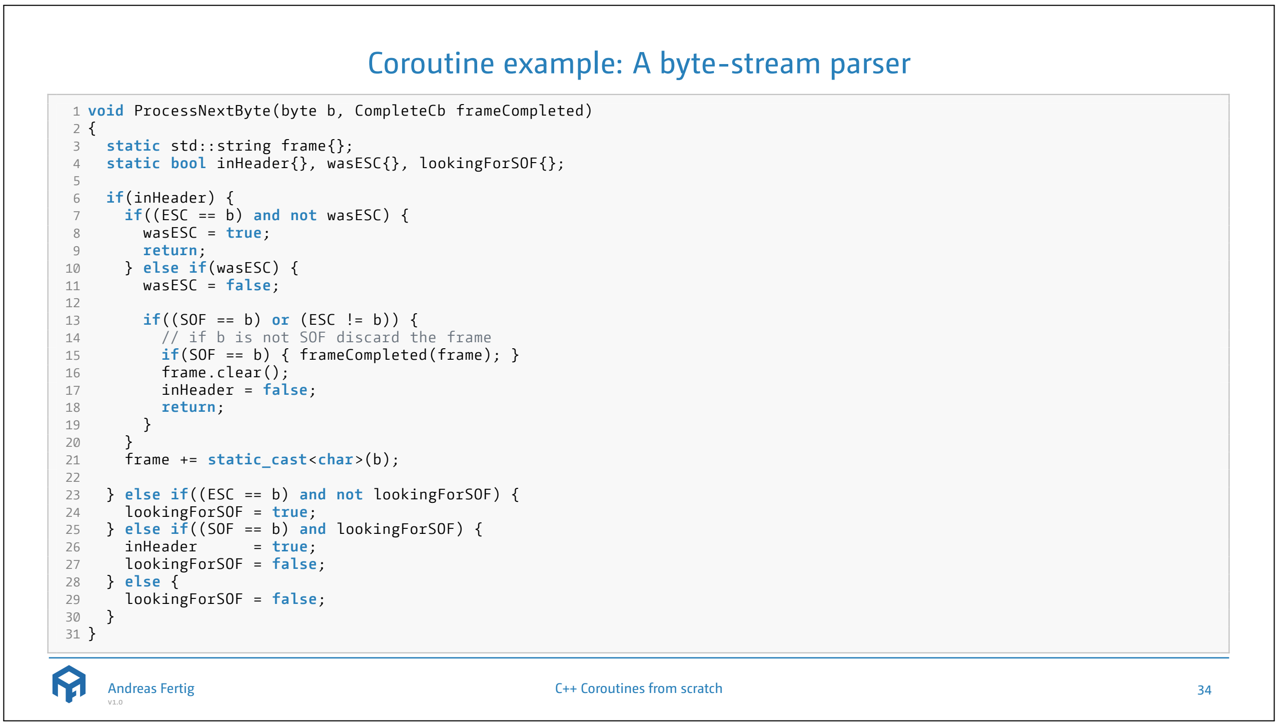


Parsing data



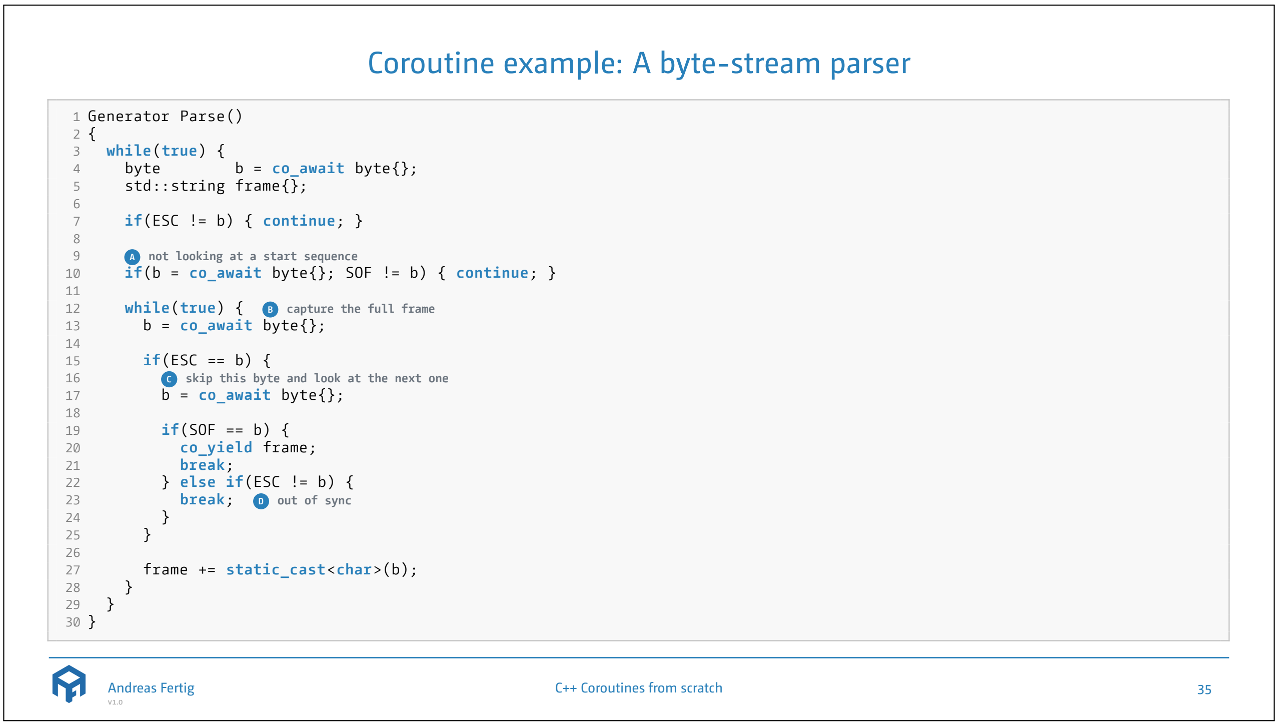
Coroutine example: A byte-stream parser

- Let’s look at a byte-stream parser
- The protocol
  - ESC (`'H'`): Escape special bytes (commands) in the stream.
  - SOF (`0x10`) start of frame: Marks the beginning of a frame.
  - ESC + SOF mark the start of a frame.



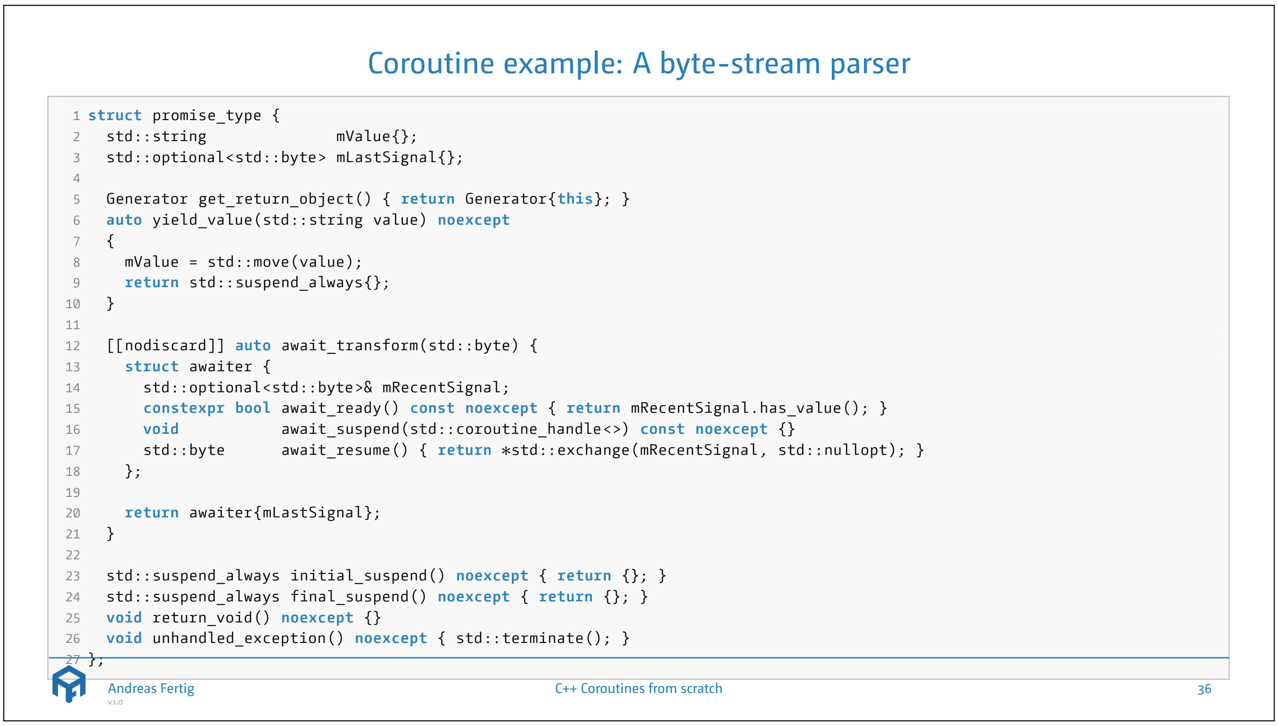
Coroutine example: A byte-stream parser

```
void ProcessNextByte(byte b, CompleteCb frameCompleted)
{
  static std::string frame{};
  static bool inHeader{}, wasESC{}, lookingForSOF{};

  if(inHeader) {
    if((ESC == b) and not wasESC) {
      wasESC = true;
      return;
    } else if(wasESC) {
      wasESC = false;

      if((SOF == b) or (ESC != b)) {
        // if b is not SOF discard the frame
        if(SOF == b) { frameCompleted(frame); }
        frame.clear();
        inHeader = false;
        return;
      }
    }
    frame += static_cast<char>(b);

  } else if((ESC == b) and not lookingForSOF) {
    lookingForSOF = true;
  } else if((SOF == b) and lookingForSOF) {
    inHeader      = true;
    lookingForSOF = false;
  } else {
    lookingForSOF = false;
  }
}
```



Coroutine example: A byte-stream parser

```
Generator Parse()
{
  while(true) {
    byte        b = co_await byte{};
    std::string frame{};

    if(ESC != b) { continue; }

    A not looking at a start sequence
    if(b = co_await byte{}; SOF != b) { continue; }

    while(true) {  B capture the full frame
      b = co_await byte{};

      if(ESC == b) {
        C skip this byte and look at the next one
        b = co_await byte{};

        if(SOF == b) {
          co_yield frame;
          break;
        } else if(ESC != b) {
          break;  D out of sync
        }
      }

      frame += static_cast<char>(b);
    }
  }
}
```



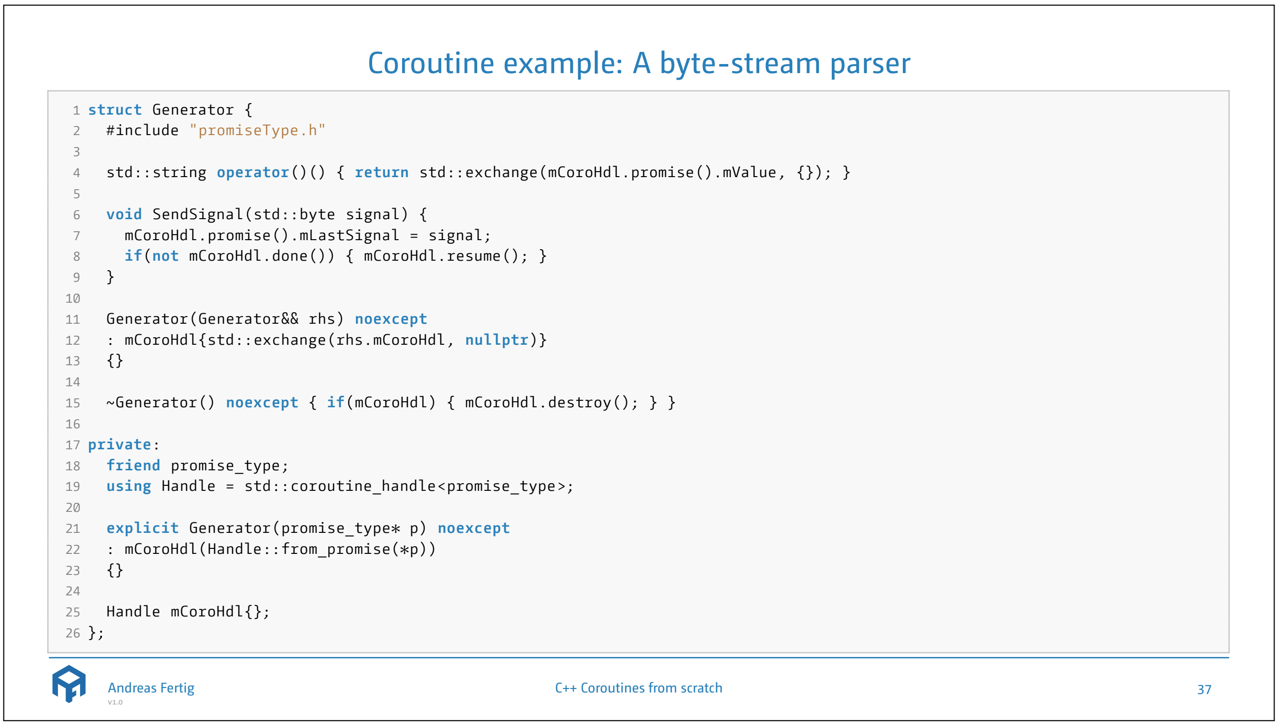
Coroutine example: A byte-stream parser

```cpp
struct promise_type {
  std::string              mValue{};
  std::optional<std::byte> mLastSignal{};

  Generator get_return_object() { return Generator{this}; }
  auto yield_value(std::string value) noexcept
  {
    mValue = std::move(value);
    return std::suspend_always{};
  }

  [[nodiscard]] auto await_transform(std::byte) {
    struct awaiter {
      std::optional<std::byte>& mRecentSignal;
      constexpr bool await_ready() const noexcept { return mRecentSignal.has_value(); }
      void           await_suspend(std::coroutine_handle<>) const noexcept {}
      std::byte      await_resume() { return *std::exchange(mRecentSignal, std::nullopt); }
    };

    return awaiter{mLastSignal};
  }

  std::suspend_always initial_suspend() noexcept { return {}; }
  std::suspend_always final_suspend() noexcept { return {}; }
  void return_void() noexcept {}
  void unhandled_exception() noexcept { std::terminate(); }
};
```

Andreas Fertig
v1.0

C++ Coroutines from scratch

36



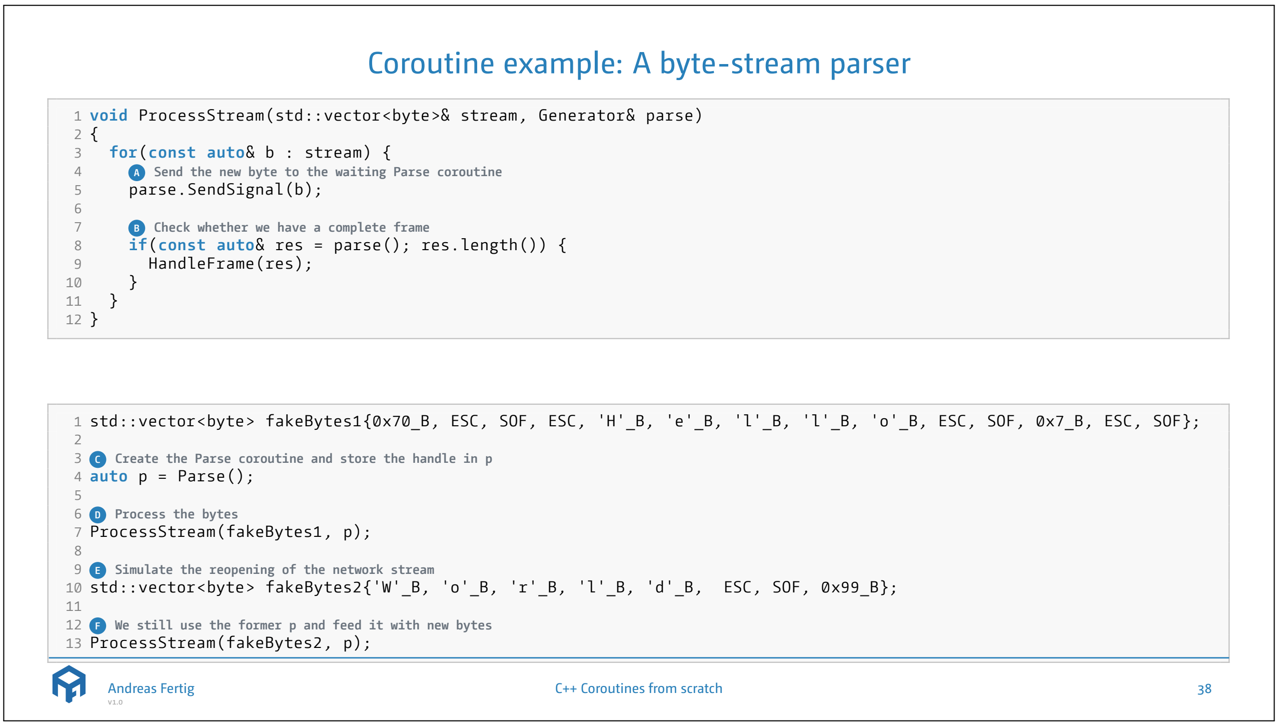
Coroutine example: A byte-stream parser

```cpp
struct Generator {
  #include "promiseType.h"

  std::string operator()() { return std::exchange(mCoroHdl.promise().mValue, {}); }

  void SendSignal(std::byte signal) {
    mCoroHdl.promise().mLastSignal = signal;
    if(not mCoroHdl.done()) { mCoroHdl.resume(); }
  }

  Generator(Generator&& rhs) noexcept
  : mCoroHdl{std::exchange(rhs.mCoroHdl, nullptr)}
  {}

  ~Generator() noexcept { if(mCoroHdl) { mCoroHdl.destroy(); } }

private:
  friend promise_type;
  using Handle = std::coroutine_handle<promise_type>;

  explicit Generator(promise_type* p) noexcept
  : mCoroHdl(Handle::from_promise(*p))
  {}

  Handle mCoroHdl{};
};
```

Andreas Fertig
v1.0

C++ Coroutines from scratch

37

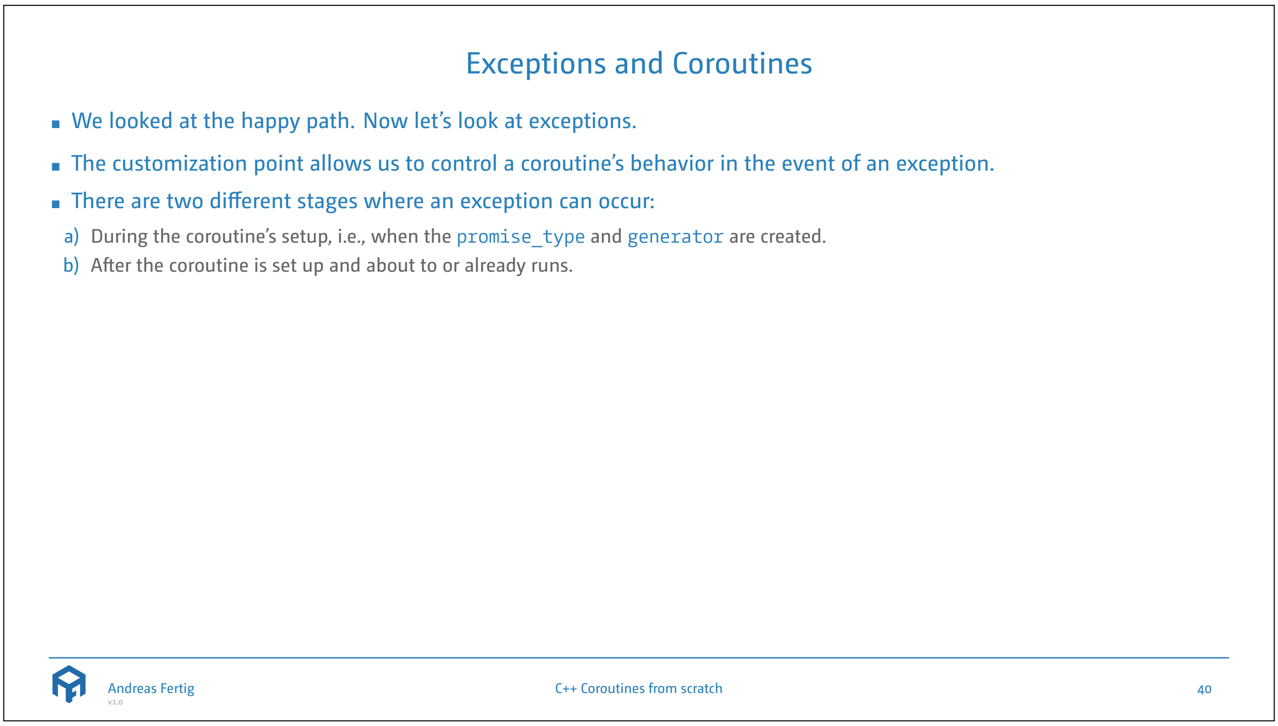


Coroutine example: A byte-stream parser

```cpp
void ProcessStream(std::vector<byte>& stream, Generator& parse)
{
  for(const auto& b : stream) {
    A Send the new byte to the waiting Parse coroutine
    parse.SendSignal(b);

    B Check whether we have a complete frame
    if(const auto& res = parse(); res.length()) {
      HandleFrame(res);
    }
  }
}
```

```cpp
std::vector<byte> fakeBytes1{0x70_B, ESC, SOF, ESC, 'H'_B, 'e'_B, 'l'_B, 'l'_B, 'o'_B, ESC, SOF, 0x7_B, ESC, SOF};

C Create the Parse coroutine and store the handle in p
auto p = Parse();

D Process the bytes
ProcessStream(fakeBytes1, p);

E Simulate the reopening of the network stream
std::vector<byte> fakeBytes2{'W'_B, 'o'_B, 'r'_B, 'l'_B, 'd'_B,  ESC, SOF, 0x99_B};

F We still use the former p and feed it with new bytes
ProcessStream(fakeBytes2, p);
```

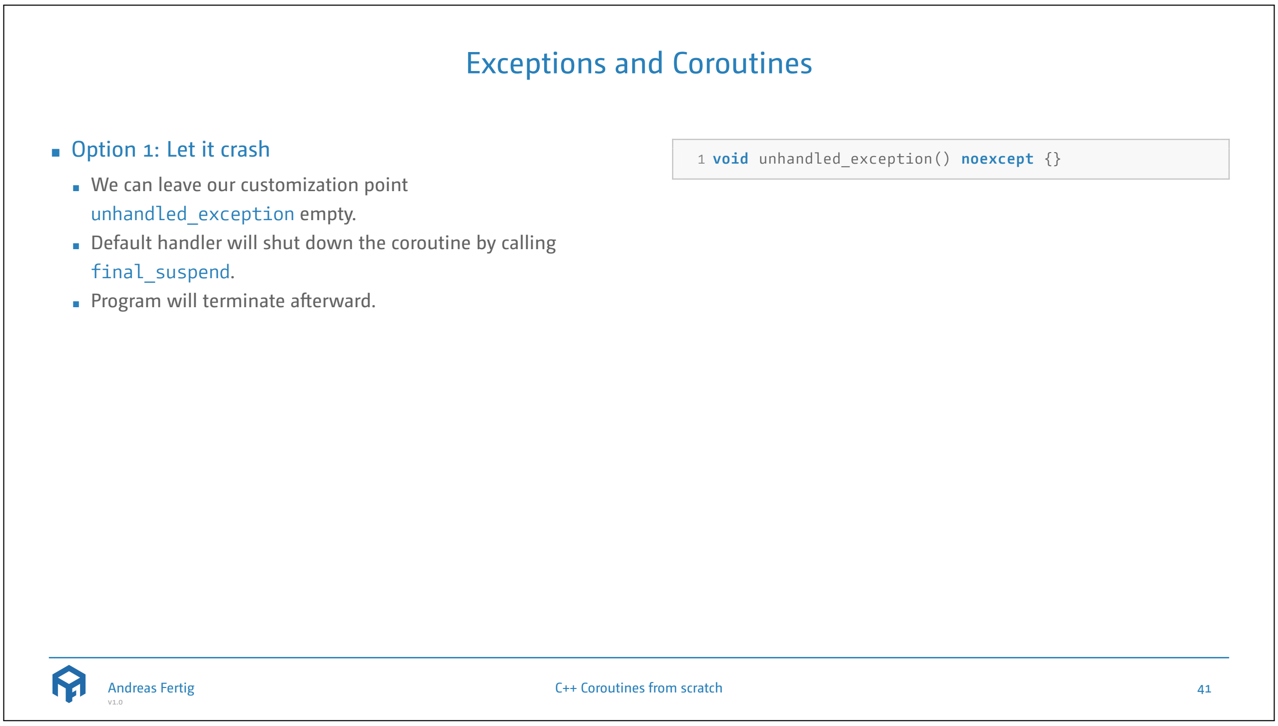
What about... exceptions?



Exceptions and Coroutines

- We looked at the happy path. Now let's look at exceptions.
- The customization point allows us to control a coroutine's behavior in the event of an exception.
- There are two different stages where an exception can occur:

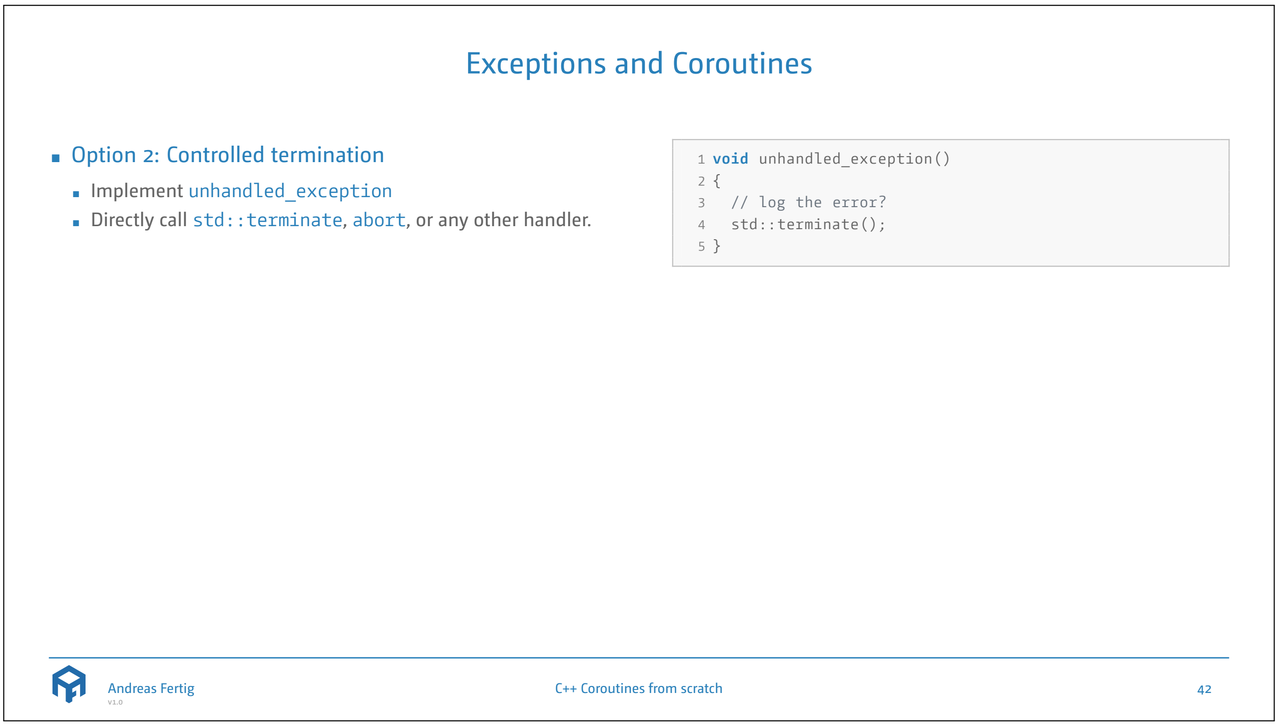
a) During the coroutine's setup, i.e., when the `promise_type` and `generator` are created.
b) After the coroutine is set up and about to or already runs.



Exceptions and Coroutines

- Option 1: Let it crash
  - We can leave our customization point `unhandled_exception` empty.
  - Default handler will shut down the coroutine by calling `final_suspend`.
  - Program will terminate afterward.

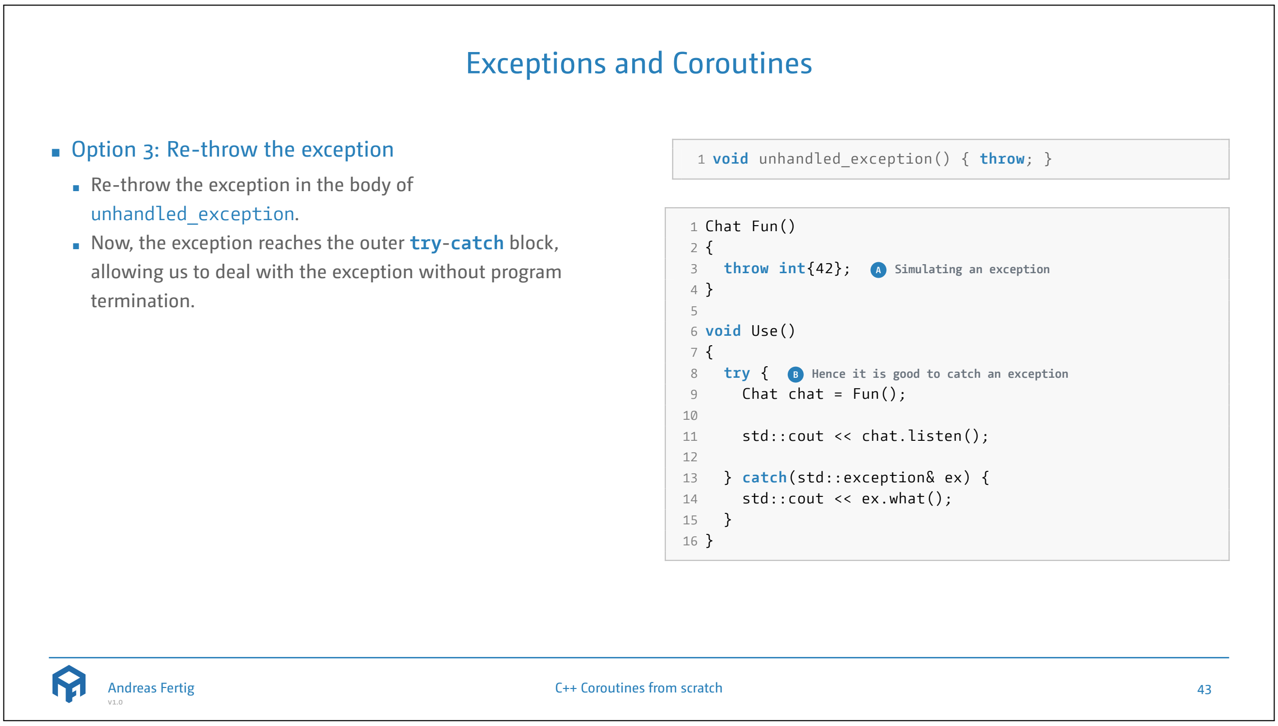
```cpp
void unhandled_exception() noexcept {}
```



Exceptions and Coroutines

- Option 2: Controlled termination
  - Implement `unhandled_exception`
  - Directly call `std::terminate`, `abort`, or any other handler.

```cpp
void unhandled_exception()
{
  // log the error?
  std::terminate();
}
```



Exceptions and Coroutines

- Option 3: Re-throw the exception
  - Re-throw the exception in the body of `unhandled_exception`.
  - Now, the exception reaches the outer **`try-catch`** block, allowing us to deal with the exception without program termination.

```cpp
void unhandled_exception() { throw; }
```

```cpp
Chat Fun()
{
  throw int{42};  Ⓐ Simulating an exception
}

void Use()
{
  try {  Ⓑ Hence it is good to catch an exception
    Chat chat = Fun();

    std::cout << chat.listen();

  } catch(std::exception& ex) {
    std::cout << ex.what();
  }
}
```

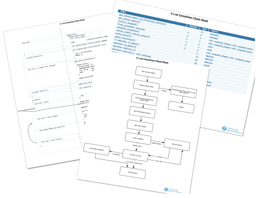
Coroutine restrictions

- There are some limitations in which functions can be a coroutine and what they have to look like.
  - `constexpr`-functions cannot be coroutines. Subsequently, this is true for `consteval`-functions.
  - Neither a constructor nor a destructor can be a coroutine.
  - A function using `varargs`. A variadic function template works.
  - A function with plain `auto` as return-type or with a concept type cannot be a coroutine. `auto` with trailing return-type works.
  - Further, a coroutine cannot use plain `return`. It must be either `co_return` or `co_yield`.
  - And last but not least, `main` cannot be a coroutine.
- Lambdas, on the other hand, can be coroutines.

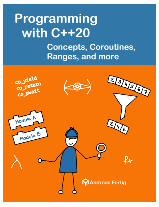
}

I am Fertig.

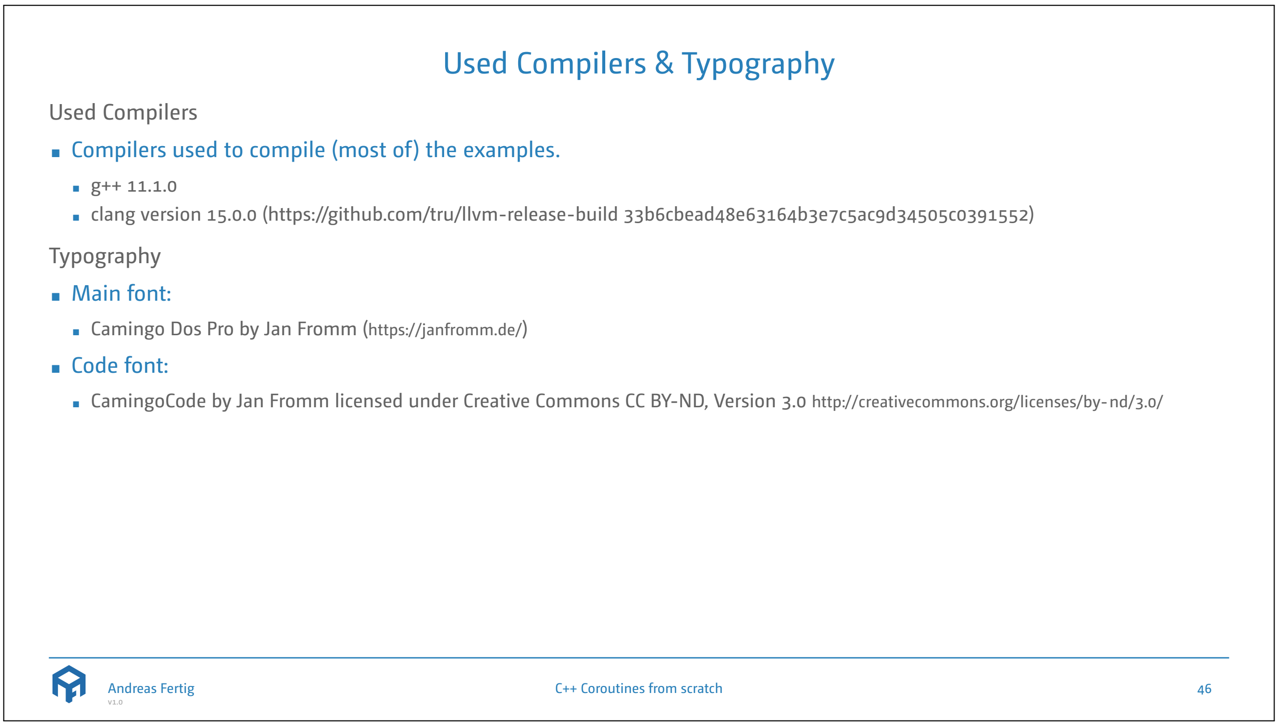
C++20 Coroutine Cheat Sheet



fertig.to/subscribe



fertig.to/bpwcpp20



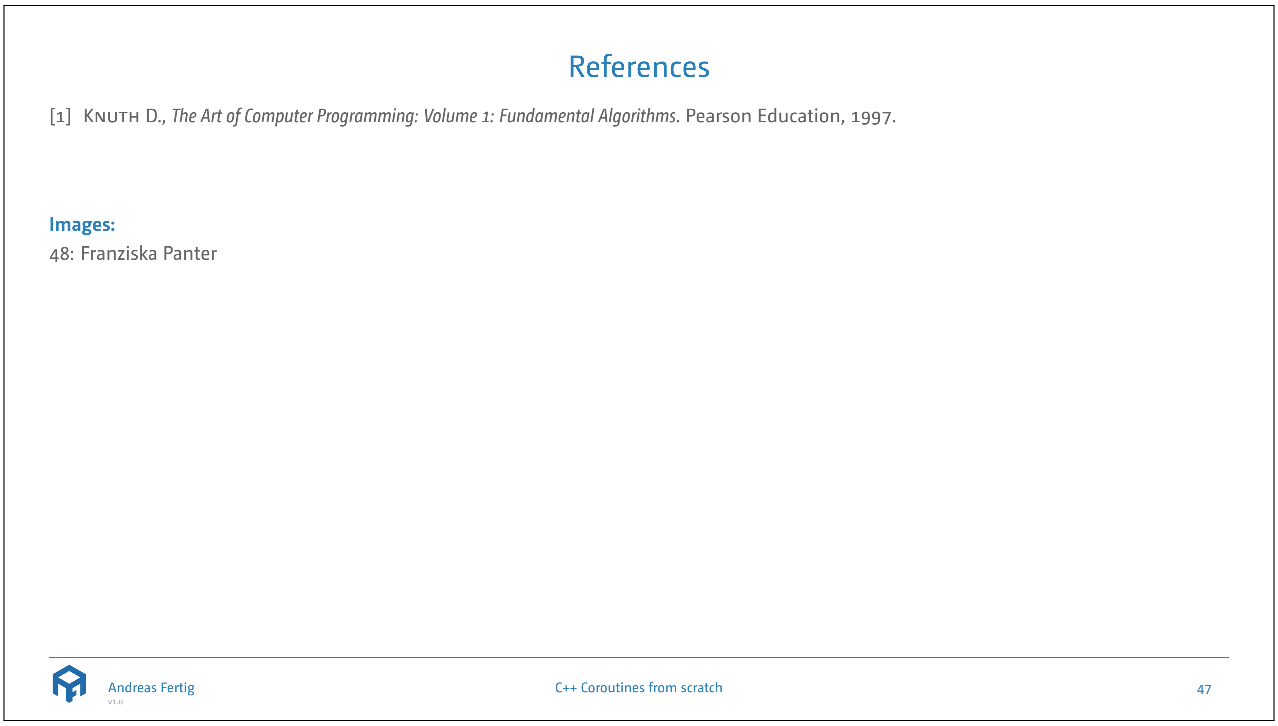
Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.
  - g++ 11.1.0
  - clang version 15.0.0 (https://github.com/tru/llvm-release-build 33b6cbead48e63164b3e7c5ac9d34505c0391552)

Typography

- Main font:
  - Camingo Dos Pro by Jan Fromm (https://janfromm.de/)
- Code font:
  - CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 http://creativecommons.org/licenses/by-nd/3.0/



References

[1] Knuth D., *The Art of Computer Programming: Volume 1: Fundamental Algorithms*. Pearson Education, 1997.

Images:

48: Franziska Panter

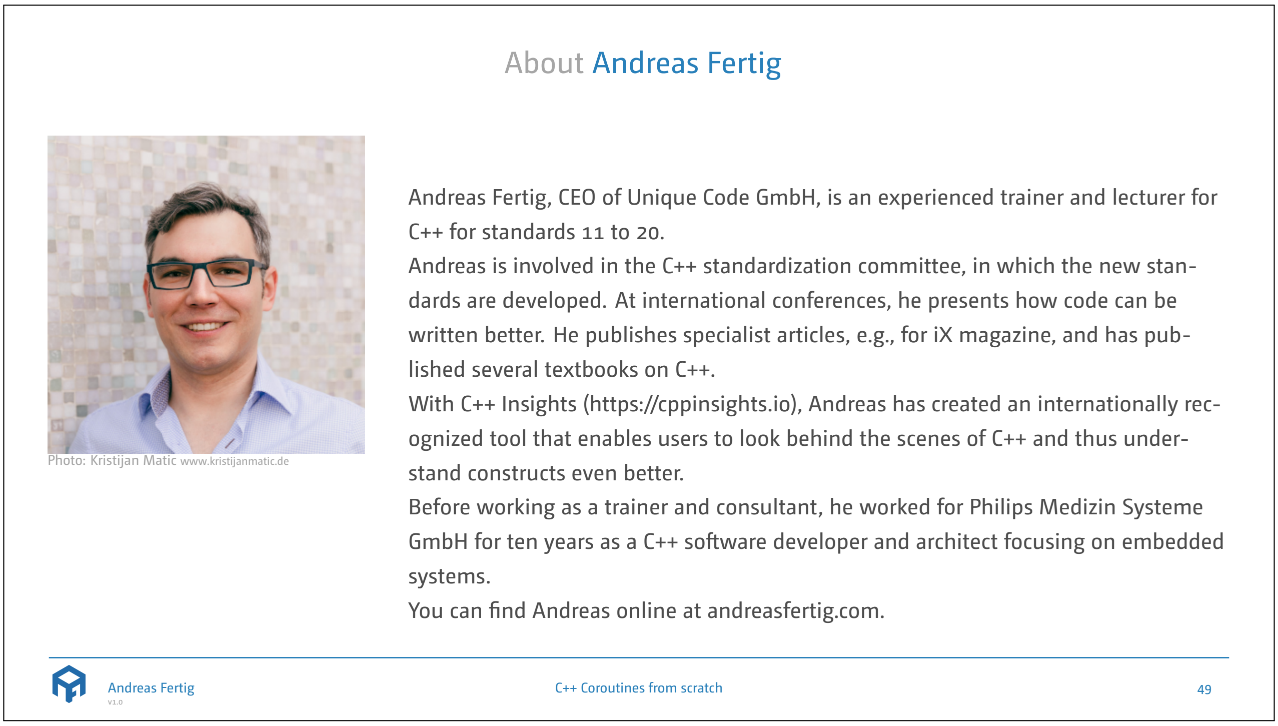
Upcoming Events

Training Classes

- *C++ Clean Code – Best Practices für Programmierer*, golem Akademie, March 27 - 29

For my upcoming talks you can check https://andreasfertig.com/talks/.
For my courses you can check https://andreasfertig.com/courses/.
Like to always be informed? Subscribe to my newsletter: https://andreasfertig.com/newsletter/.

MARK YOUR CALENDER



About Andreas Fertig

Photo: Kristijan Matic www.kristijanmatic.de

Andreas Fertig, CEO of Unique Code GmbH, is an experienced trainer and lecturer for C++ for standards 11 to 20.
Andreas is involved in the C++ standardization committee, in which the new standards are developed. At international conferences, he presents how code can be written better. He publishes specialist articles, e.g., for iX magazine, and has published several textbooks on C++.
With C++ Insights (https://cppinsights.io), Andreas has created an internationally recognized tool that enables users to look behind the scenes of C++ and thus understand constructs even better.
Before working as a trainer and consultant, he worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.
You can find Andreas online at andreasfertig.com.