# Учимся готовить C++ корутины на практике

# Understanding C++ coroutines by example

Pavel Novikov

@cpp_ape

R&D Align Technology

# No decent user facing support in C++20

Use `cppcoro` by Lewis Baker

https://github.com/lewissbaker/cppcoro

Thanks for coming!

# Gameplan

- Iteration 0: my first coroutine
  - What is a C++ coroutine?
  - Demystifying compiler magic
- Iteration 1: awaiting tasks
  - Making tasks awaitable
  - Writing awaitable types
- Iteration 2:
  - Getting tasks result
  - Thread safety
- Analysis of the approach

# Iteration 0: my first coroutine

```
Task<int> foo() {
  co_return 42;
}
```

A function is a coroutine if it contains one of these:

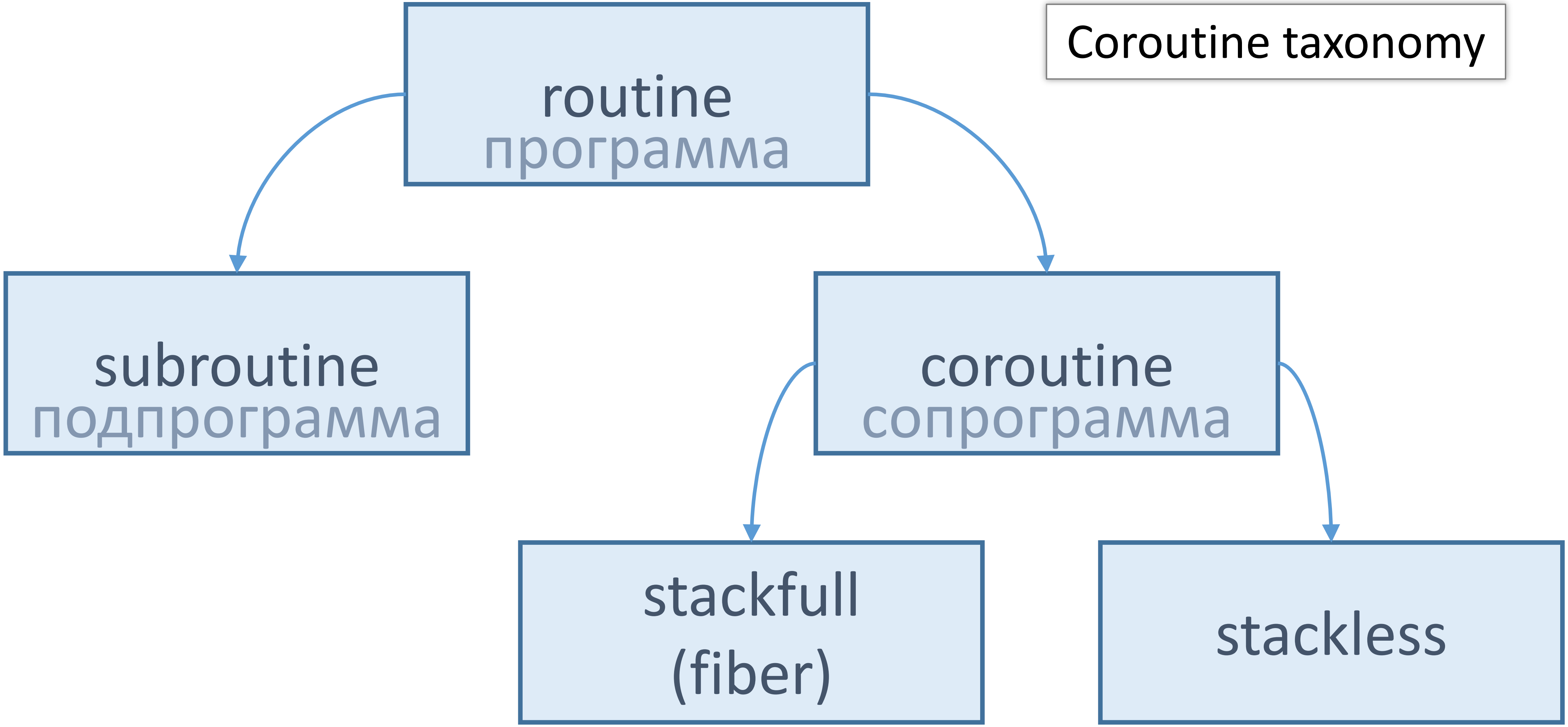
`co_return` (coroutine return statement)

`co_await` (await expression)

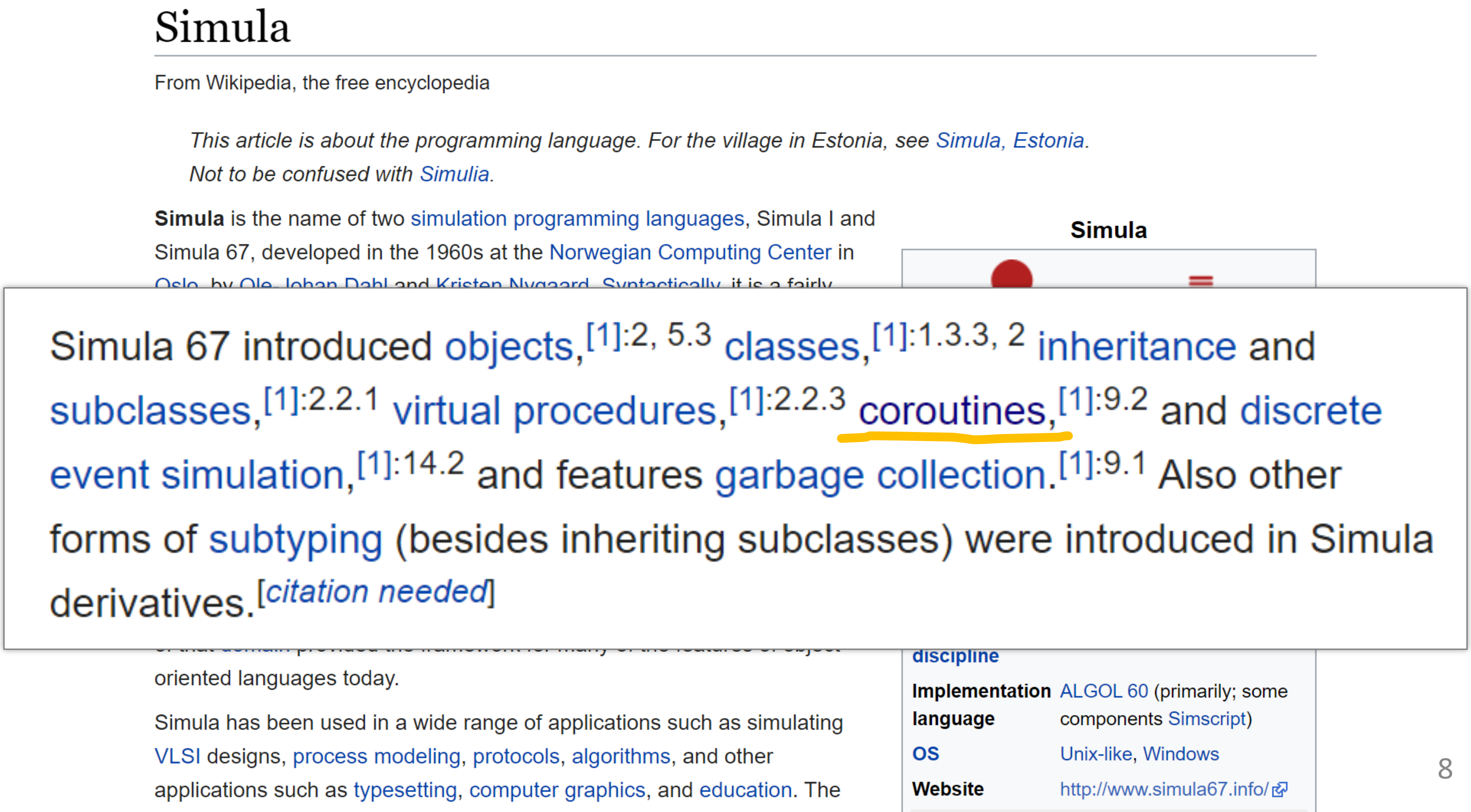
`co_yield` (yield expression)

# What is a C++ coroutine?

Coroutine taxonomy



# What is a C++ coroutine?



## Simula

From Wikipedia, the free encyclopedia

*This article is about the programming language. For the village in Estonia, see Simula, Estonia.*

*Not to be confused with Simulia.*

**Simula** is the name of two simulation programming languages, Simula I and Simula 67, developed in the 1960s at the Norwegian Computing Center in Oslo, by Ole-Johan Dahl and Kristen Nygaard. Syntactically, it is a fairly

Simula 67 introduced objects,[1]:2, 5.3 classes,[1]:1.3.3, 2 inheritance and subclasses,[1]:2.2.1 virtual procedures,[1]:2.2.3 coroutines,[1]:9.2 and discrete event simulation,[1]:14.2 and features garbage collection.[1]:9.1 Also other forms of subtyping (besides inheriting subclasses) were introduced in Simula derivatives.[*citation needed*]

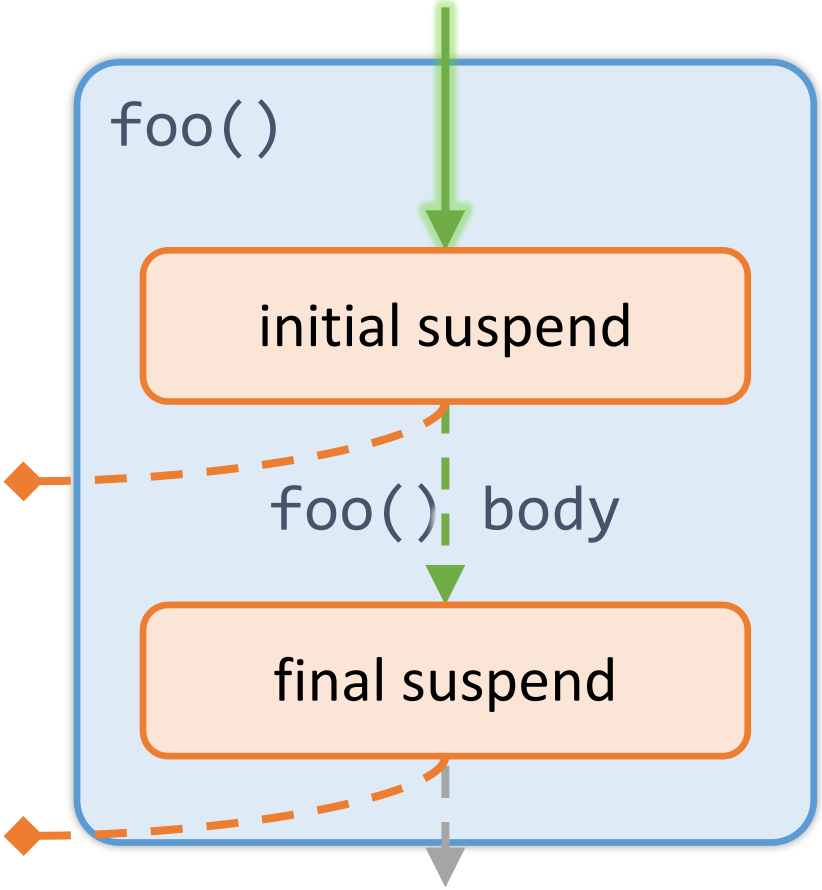
of that domain provided the framework for many of the features of object oriented languages today.

Simula has been used in a wide range of applications such as simulating VLSI designs, process modeling, protocols, algorithms, and other applications such as typesetting, computer graphics, and education. The

| **Simula** | |
|---|---|
| **discipline** | |
| **Implementation language** | ALGOL 60 (primarily; some components Simscript) |
| **OS** | Unix-like, Windows |
| **Website** | http://www.simula67.info/ |

# What is a C++ coroutine?

```cpp
Task<int> foo() {
  co_return 42;
}
```



A coroutine behaves as if its *function-body* were replaced by:

```
{
  promise-type promise promise-constructor-arguments ;
  try {
    co_await promise.initial_suspend() ;
    function-body
  } catch ( ... ) {
    if (!initial-await-resume-called)
      throw ;
    promise.unhandled_exception() ;
  }
final-suspend :
  co_await promise.final_suspend() ;
}
```

# What is a C++ coroutine?

```
Task<int> foo() {
  co_return 42;
}
```

A coroutine behaves as if its *function-body* were replaced by:

```
{
  promise-type promise promise-constructor-arguments ;
  try {
    co_await promise.initial_suspend() ;
    function-body
  } catch ( ... ) {
    if (!initial-await-resume-called)
      throw ;
    promise.unhandled_exception() ;
  }
final-suspend :
  co_await promise.final_suspend() ;
}
```

# Transformation by the compiler

```
Task<int> foo() {
  co_return 42;
}
```

original code

transformed code

```
Task<int> foo() {
  struct CoroFrame {
    Task<int>::promise_type promise;
    bool initial_await_resume_called = false;
    int state = 0;
    void operator()() {


        co_return 42;









    }
  };
  auto coroFrame = new CoroFrame;
  auto returnObject{ coroFrame->promise.get_return_object() };
  (*coroFrame)();
  return returnObject;
}
```

# Transformation by the compiler

```
Task<int> foo() {
  co_return 42;
```

```
Task<int> foo() {
  struct CoroFrame {
    Task<int>::promise_type promise;
    bool initial_await_resume_called = false;
    int state = 0;
    void operator()() {
      try {
```

```
ct() };
```

```
  return returnObject;
}
```

```
Task<int> foo() {
  struct CoroFrame {
    Task<int>::promise_type promise;
    bool initial_await_resume_called = false;
    int state = 0;
    void operator()() { /*...*/ }
  };
  auto coroFrame = new CoroFrame;
  auto returnObject{ coroFrame->promise.get_return_object() };
  (*coroFrame)();
  return returnObject;
}
```

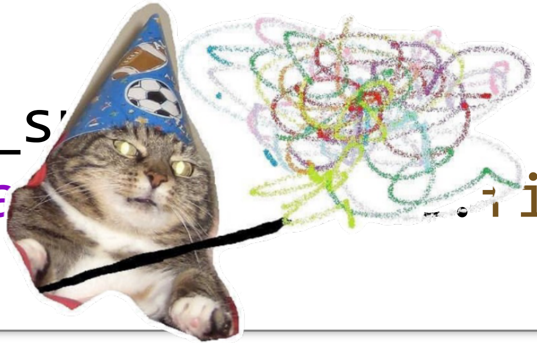
coroutine frame

# Transformation by the compiler

```
Task<int> foo() {
  co_return 42;
}
```

```
Task<int> foo() {
  struct CoroFrame {
    Task<int>::promise_type promise;
    bool initial_await_resume_called = false;
    int state = 0;
    d operator()() {
```

```
void ope
  try {
    co_a       promise.initial_suspend();
    co_return 42;
  }
  catch (...) {
    if (!initial_await_resume_called)
      throw;
    promise.unhandled_exception();
  }
final_s
  co_a             inal_suspend();
}
```



```
ject() };
}
```

# Transformation by the compiler

```
Task<int> foo() {
  co_return 42;
}
```

```
Task<int> foo() {
  struct CoroFrame {
    Task<int>::promise_type promise;
    bool initial_await_resume_called = false;
    int state = 0;
    void operator()() {
```

```
                                                        ject() };
```

```
}
```

```
void operator()() {
  try {
    co_await promise.initial_suspend();
    promise.return_value(42); goto final_suspend;
  }
  catch (...) {
    if (!initial_await_resume_called)
      throw;
    promise.unhandled_exception();
  }
final_suspend:
  co_await promise.final_suspend();
}
```

# Transformation by the compiler

```
Task<int> foo() {
  co_return 42;
}
```

Sequence of operations:

```
Task<int>::promise_type promise;
promise.get_return_object();
  promise.initial_suspend();
  promise.return_value(42);
  promise.unhandled_exception();
  promise.final_suspend();
```

```
Task<int> foo() {
  struct CoroFrame {
    Task<int>::promise_type promise;
    bool initial_await_resume_called = false;
    int state = 0;
    void operator()() {
      try {
        co_await promise.initial_suspend();
        promise.return_value(42); goto final_suspend;
      }
      catch (...) {
        if (!initial_await_resume_called)
          throw;
        promise.unhandled_exception();
      }
    final_suspend:
      co_await promise.final_suspend();
    }
  };
  auto coroFrame = new CoroFrame;
  auto returnObject{ coroFrame->promise.get_return_object() };
  (*coroFrame)();
  return returnObject;
}
```

# Task type

```
template<typename T> struct Promise;
```

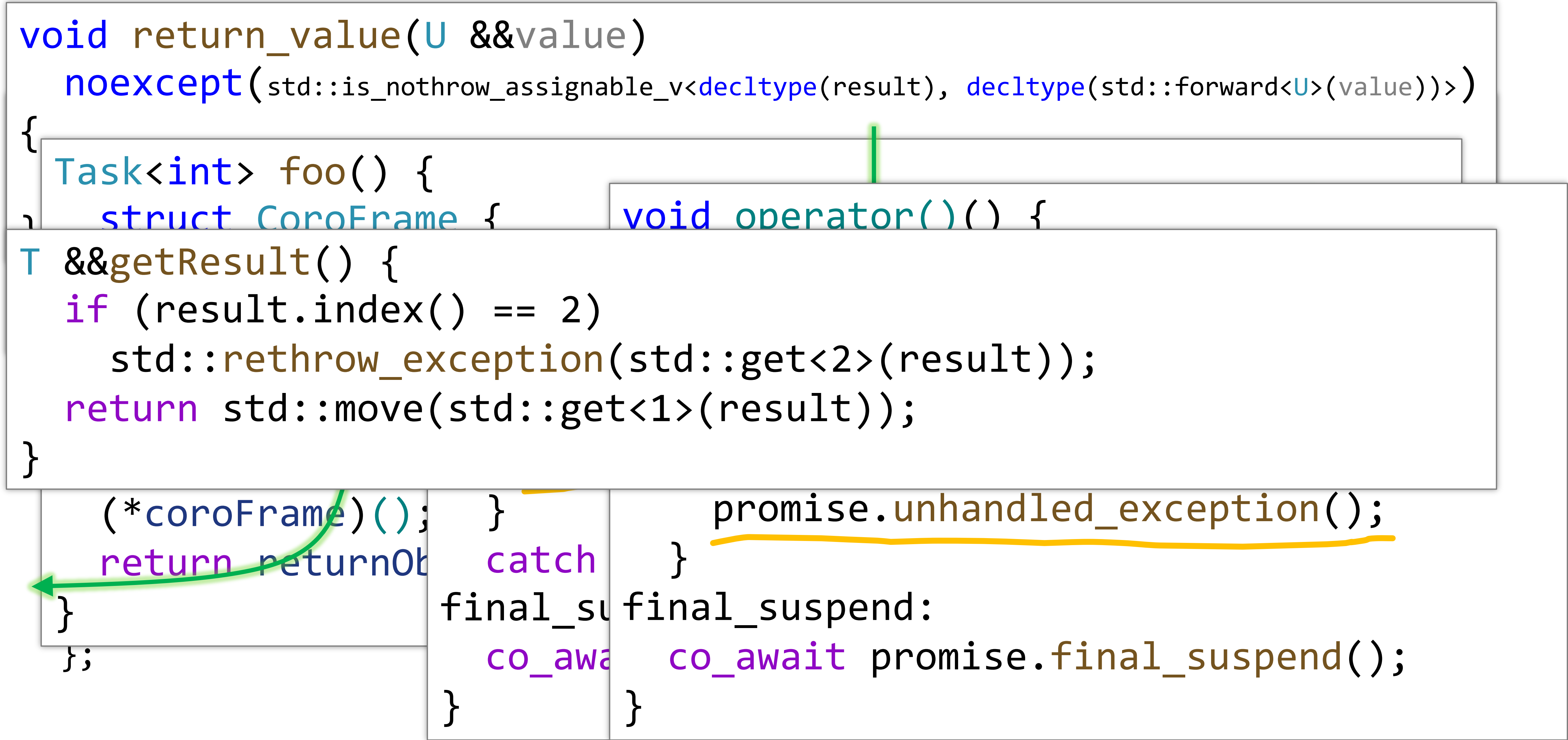
```
struct CoroDeleter {
  template<typename Promise>
  void operator()(Promise *promise) const noexcept {
    using CoroHandle = std::coroutine_handle<Promise>;
    CoroHandle::from_promise(*promise).destroy();
  }
};
template<typename T>
using PromisePtr = std::unique_ptr<Promise<T>, CoroDeleter>;
```

```
  PromisePtr<T> promise = nullptr;

  template<typename> friend struct Promise;
};
```

# Task type

```
template<typename T> struct Promise;

template<typename T>
struct [[nodiscard]] Task {
  using promise_type = Promise<T>;
  Task() = default;

private:
  Task(Promise<T> *promise) : promise{ promise } {}

  PromisePtr<T> promise = nullptr;

  template<typename> friend struct Promise;
};
```

# Promise type



```
void return_value(U &&value)
  noexcept(std::is_nothrow_assignable_v<decltype(result), decltype(std::forward<U>(value))>)
{
}
```

```
Task<int> foo() {
  struct CoroFrame {
    (*coroFrame)();
    return returnOb
}
};
```

```
  }
  catch
final_su
  co_awa
}
```

```
void operator()() {
    promise.unhandled_exception();
  }
final_suspend:
  co_await promise.final_suspend();
}
```

```
T &&getResult() {
  if (result.index() == 2)
    std::rethrow_exception(std::get<2>(result));
  return std::move(std::get<1>(result));
}
```

# Iteration 0: my first coroutine

```
Task<int> foo() {
  std::cout << "foo(): about to return\n";
  co_return 42;
}

auto task = foo();
```

output:

```
foo(): about to return
```

# Iteration 0: my first coroutine

```
Task<void> foo
  co_return;
}

template<typen
struct Promise
  //...
  void return_
  //...
};
```

```
ial_suspend();
; goto final_suspend;


suspend();
```

# Iteration 1: awaiting tasks

```
Task<int> bar() {
  const auto result = foo();
  const int i = co_await result;
  co_return i + 23;
}
```

# Awaiting: rough idea

```
co_await result;
```

```
  auto awaitable{ getAwaitable(result) };
  if (!awaitable.await_ready()) {
    awaitable.await_suspend(thisCoroHandle);
    // suspend coroutine
  }
resume:
  awaitable.await_resume();
```

# Transformation by the compiler

```
Task<int> bar() {
  const auto result = foo();
  const int i = co_await result;
  co_return i + 23;
}
```

transformed code

original code

```
Task<int> bar() {
  struct CoroFrame {
    Task<int>::promise_type promise;
    bool initial_await_resume_called = false;
    int state = 0;
    //...
    void operator()();
  };
  auto coroFrame = new CoroFrame;
  auto returnObject{
    coroFrame->promise.get_return_object()
  };
  (*coroFrame)();
  return returnObject;
}
```

# Transformation by the compiler

```
void operator()() {
  try {
    switch (state)
    {
    case 0:
      break;
    case 1:
      goto initialResume;
    case 2:
      goto resume2;
    default:
      break; //bad ☹
    }
  //...
```

```
struct CoroFrame {
  Task<int>::promise_type promise;
  bool initial_await_resume_called = false;
  int state = 0;
  //...
  void operator()();
};
```

# Transformation by the compiler

```
void operator()() {
  //...
    state = 1;
    awaitable0 ← ???
    if (!awaitable0->aw
      awaitable0->await
      // suspend
      return;
    }
  initialResume:
    initial_await_resume_called = true;
    awaitable0->await_resume();
  //...
```

```
struct CoroFrame {
  Task<int>::promise_type promise;
  bool initial_await_resume_called = false;
  int state = 0;
  std::optional<Awaitable0> awaitable0;
  //...
  void operator()();
};
```

# Transformation by the compiler

```
co_await promise.initial_suspend();
```

```
void operator()() {
  //...
    state = 1;
    awaitable0.emplace(getAwaitable(promise.initial_suspend()));
    if (!awaitable0->await_ready()) {
      awaitable0->await_suspend(thisCoroHandle);

  ir


  //
```

```
struct suspend_never {
  bool await_ready() noexcept {
    return true;
  }
  void await_suspend(coroutine_handle<>) noexcept {}
  void await_resume() noexcept {}
};
```

# Transformation by the compiler

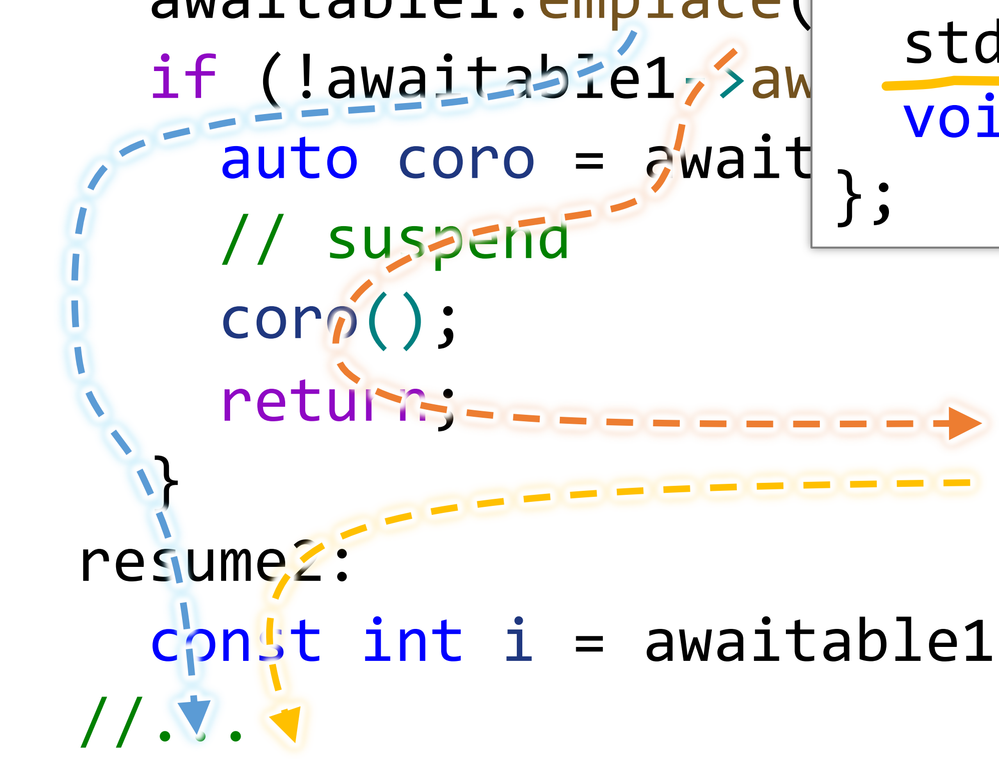
```
void operator()() {
  //...
    const auto result = foo();
    state = 2;
    awaitable1.emplace(getAwaitable(result));
    if (!awaitable1->await_ready() {
      auto coro = awaitable1->await_suspend(thisCoroHandle);
  //...
```

```
const auto result = foo();
const int i = co_await result;
```

current coroutine to suspend

```
struct Awaitable {
  bool await_ready() const noexcept;
  using CoroHandle = std::coroutine_handle<>;
  CoroHandle await_suspend(CoroHandle) const noexcept;
  T &&await_resume() const;
};
```

# Transformation by the compiler

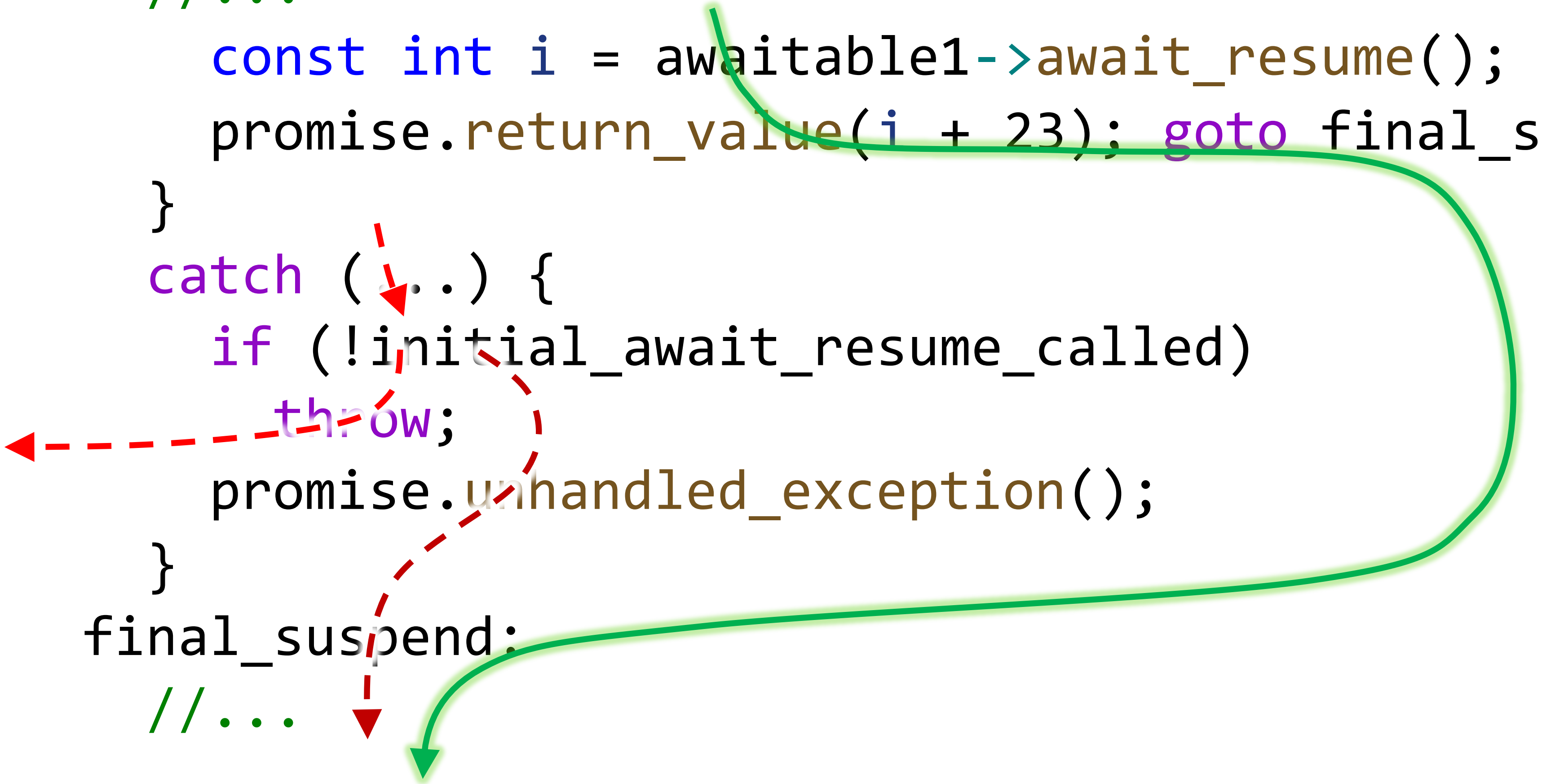


```
void operator()() {
  //...
    const auto result =
    state = 2;
    awaitable1.emplace(
    if (!awaitable1->aw
      auto coro = await
      // suspend
      coro();
      return;
    }
  resume2:
    const int i = awaitable1->await_resume();
  //...
```

```
struct CoroFrame {
  Task<int>::promise_type promise;
  bool initial_await_resume_called = false;
  int state = 0;
  std::optional<Awaitable0> awaitable0;
  std::optional<Awaitable1> awaitable1;
  void operator()();
};
```

# Transformation by the compiler

```
co_return i + 23;
```



```
void operator()() {
 //...
  const int i = awaitable1->await_resume();
  promise.return_value(i + 23); goto final_suspend;
 }
 catch (...) {
  if (!initial_await_resume_called)
   throw;
  promise.unhandled_exception();
 }
final_suspend:
 //...
```

# Transformation by the compiler

```
void operator()() {
  //...
final_suspend:
  auto finalAwaitable{ getAwaitable(promise.final_suspend()) };
  if (!finalAwaitable.await_ready()) {
    finalAwaitable.await_suspend(thisCoroHandle);

  }
  de
}
```

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

```
struct suspend_always {
  bool await_ready() noexcept {
    return false;
  }
  void await_suspend(coroutine_handle<>) noexcept {}
  void await_resume() noexcept {}
};
```

# Awaiting: Task

```
template<typename T>
struct [[nodiscard]] Task {
  using promise_type = Promise<T>;
  Task() = default;
  auto operator co_await() const noexcept;

private:
  Task(Promise<T> *promise) : promise{ promise } {}

  PromisePtr<T> *promise = nullptr;

  template<typename> friend struct Promise;
};
```

# Task::operator co_await

```
auto operator co_await() const noexcept {
  struct Awaitable {
    //...
    Promise<T> &promise;
  };
  return Awaitable{ *promise };
}
```

# Task::operator co_await

```
struct Awaitable {
  bool await_ready() const noexcept {
    return promise.isReady();
  }
  using CoroHandle = std::coroutine_handle<>;
  CoroHandle await_suspend(CoroHandle continuation) const noexcept {
    promise.continuation = continuation;
    return std::coroutine_handle<Promise<T>>::from_promise(promise);
  }
  T &&await_resume() const {
    return promise.getResult();
  }

  Promise<T> &promise;
};
```

symmetric control transfer

current coroutine is suspended and

suspended coroutine is returned and resumed

# Awaiting: Promise

```
template<typename T>
struct Promise {
  //...
  // std::suspend_always final_suspend() noexcept { return {}; }
  auto final_suspend() noexcept {
    struct FinalAwaitable { /*...*/ };
    return FinalAwaitable{};
  }
  //...
  std::variant<std::monostate, T, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

# Awaiting: Promise

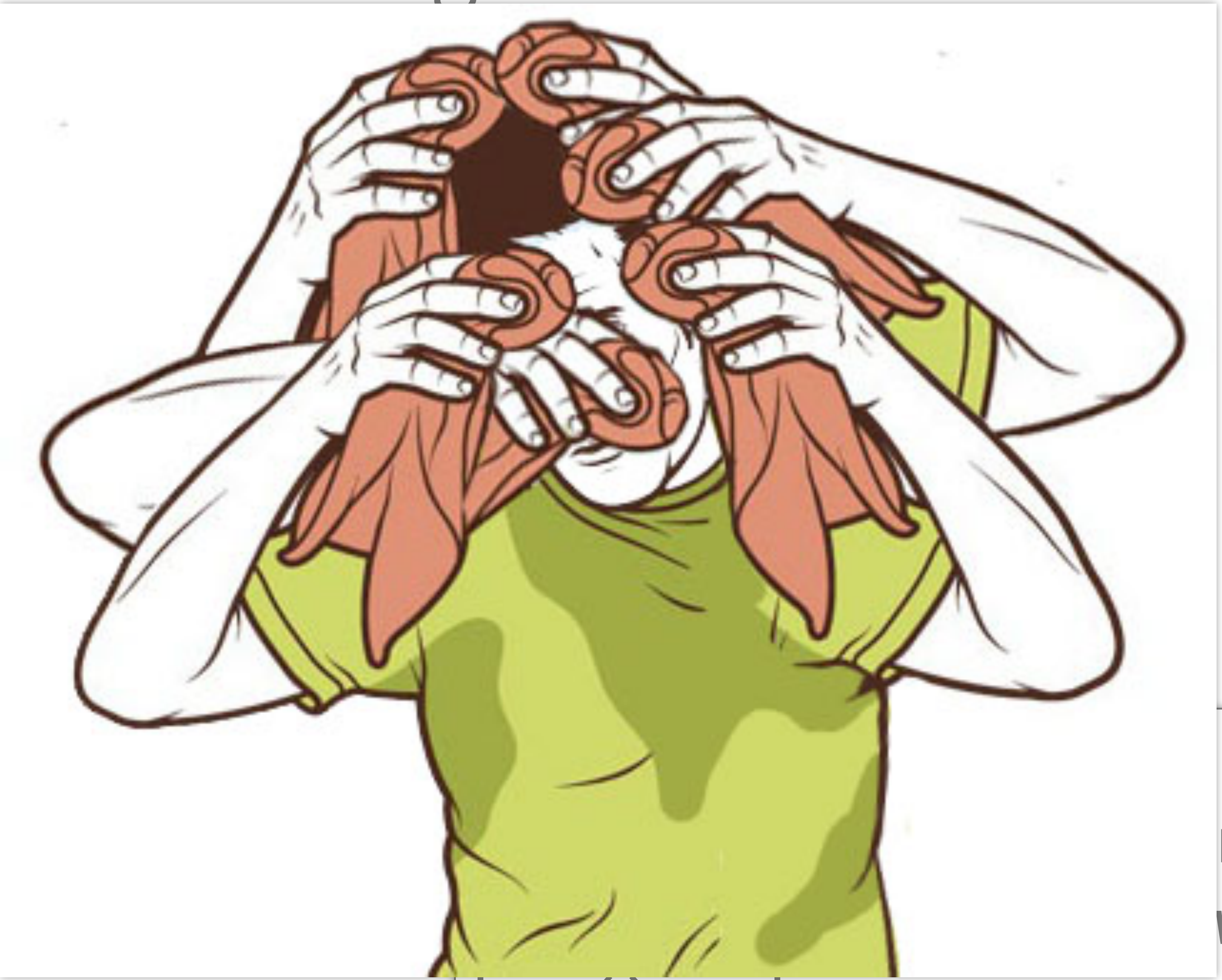
```
template<typename T>
struct Promise {
```

```
struct FinalAwaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(std::coroutine_handle<Promise<T>> thisCoro) {
    auto &promise = thisCoro.promise();
    if (promise.continuation)
      promise.continuation();
  }
  void await_resume() const noexcept {}
};
```

```
    std::coroutine_handle<> continuation;
};
```

# Iteration 1: awaiting tasks

```
Task<int>
  const au
  std::cou                                           ;
  const in
  std::cou
  co_retur
}

auto task
```



```
                          rn
                          wait
bar(): about to return
```

# Helpful tip

Write constructor and destructor for promise types.

```
template<typename T>
struct Promise {
  Promise() {
    std::cout << "Promise: ctor\n";
  }
  ~Promise() {
    std::cout << "Promise: dtor\n";
  }
  //...
```

# Writing an awaitable

```
struct Sleep {
  bool await_ready() const noexcept {
    return duration == duration.zero();
  }
  void await_suspend(std::coroutine_handle<> coro) const {
    std::this_thread::sleep_for(duration);
    coro();
  }
  void await_resume() const noexcept {}

  std::chrono::milliseconds duration;
};
```

suspended coroutine

ead to sleep

resumes the suspended coroutine

# Writing an awaitable

```
Task<void> sleepy() {
  std::
  co_aw
  std::
}
auto ta
```

```
eep
turn
Promise::dtor
```

# Asyncronously reading a file

```
struct AsyncReadFile {
  AsyncReadFile(std::filesystem::path path) :
    path{ std::move(path) } {}
  bool await_ready() const noexcept { return false; }
  void await_suspend(std::coroutine_handle<> coro);
  std::string await_resume() noexcept {
    return std::move(result);
 }

private:
  std::filesystem::path path;
  std::string result;
};
```

# Asyncronously reading a file

```
void await_suspend(std::coroutine_handle<> coro) {
  auto work = [this, coro]() mutable {
    std::cout << tid << " worker thread: opening file\n";
    std::cout << tid << " worker thread: resuming coro\n";
    coro();
    std::cout << tid << " worker thread: exiting\n";
  };
  std::thread{ work }.detach();
}
```

Clang:
no matching function for call to object of type 'const std::coroutine_handle<>'
include/c++/v1/experimental/coroutine:113:10: note: candidate function not viable:
'this' argument has type 'const std::coroutine_handle<>', but method is not marked const

# Asyncronously reading a file

```
void await_suspend(std::coroutine_handle<> coro) {
  auto work = [this, coro]() mutable {
    std::cout << tid << " worker thread: opening file\n";
                               path };
                                thread: reading file\n";
                              f_iterator<char>{stream},
                              f_iterator<char>{});
                                thread: resuming coro\n";

                                thread: exiting\n";

  std::thread{ work }.detach();
}
```

# Asyncronously reading a file

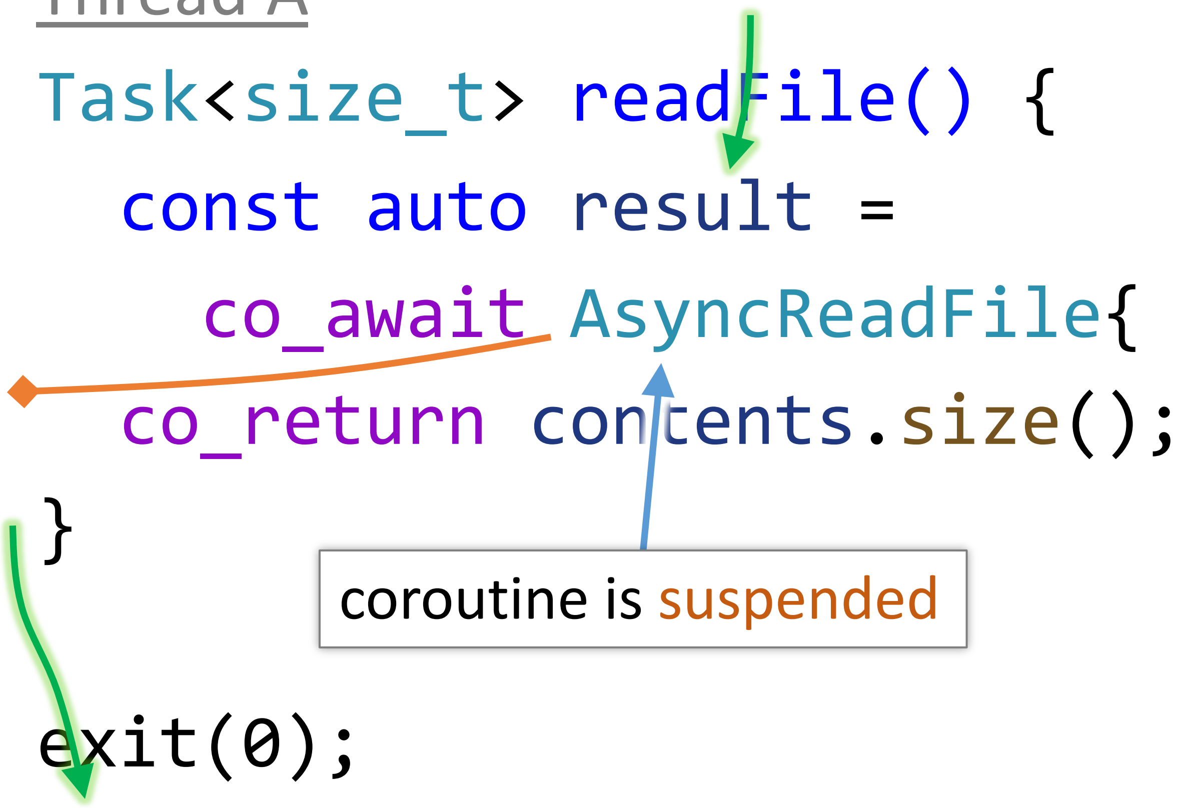
```
Task<size_t> readFile() {
  std::cout << tid << " readFile(): about to read file async\n";
  const auto result = co_await AsyncReadFile{ "main.cpp" };
  std::cout << tid << " readFile(): about to return (size "
    << result.size() << ")\n";
  co_return result.size();
}


int main() {
  auto task = readFile();
}
```

output:

```
Promise: ctor
(tid=38216) readFile(): about to read file async
Promise: dtor
```

# Asyncronously reading a file

Thread A



```
Task<size_t> readFile() {
  const auto result =
    co_await AsyncReadFile{ "main.cpp" };
  co_return contents.size();
}

exit(0);
```

coroutine is suspended

Thread B

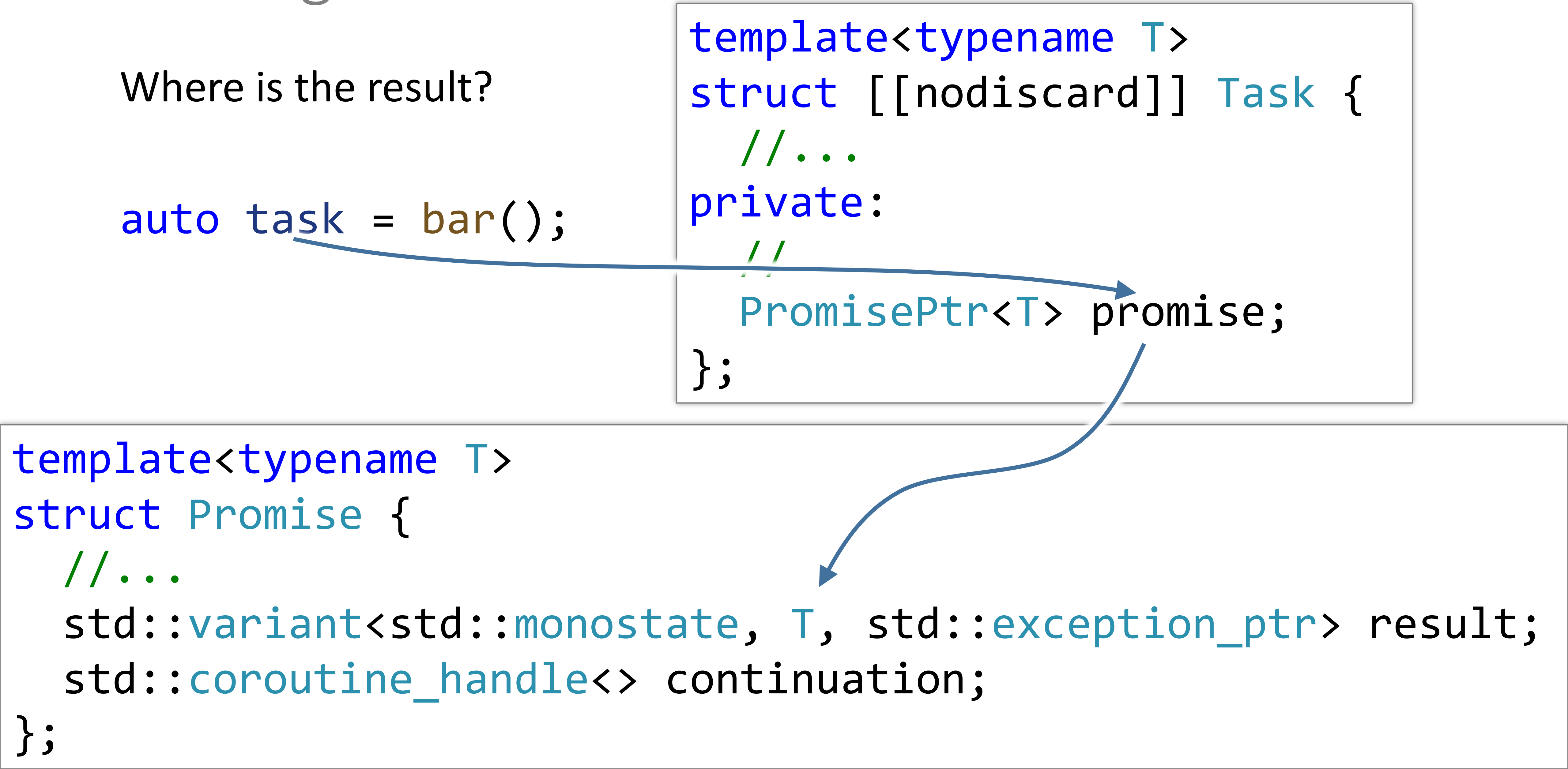
```
auto work = [this, coro]() {
  //...
  coro();
  //...
};
```

# Iteration 2

In which we learn how to get result out of a task and make awaiting thread-safeish

# Getting result from task

Where is the result?



```
auto task = bar();
```

```
template<typename T>
struct [[nodiscard]] Task {
  //...
private:
  //
  PromisePtr<T> promise;
};
```

```
template<typename T>
struct Promise {
  //...
  std::variant<std::monostate, T, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

# Getting result from task

Thread A

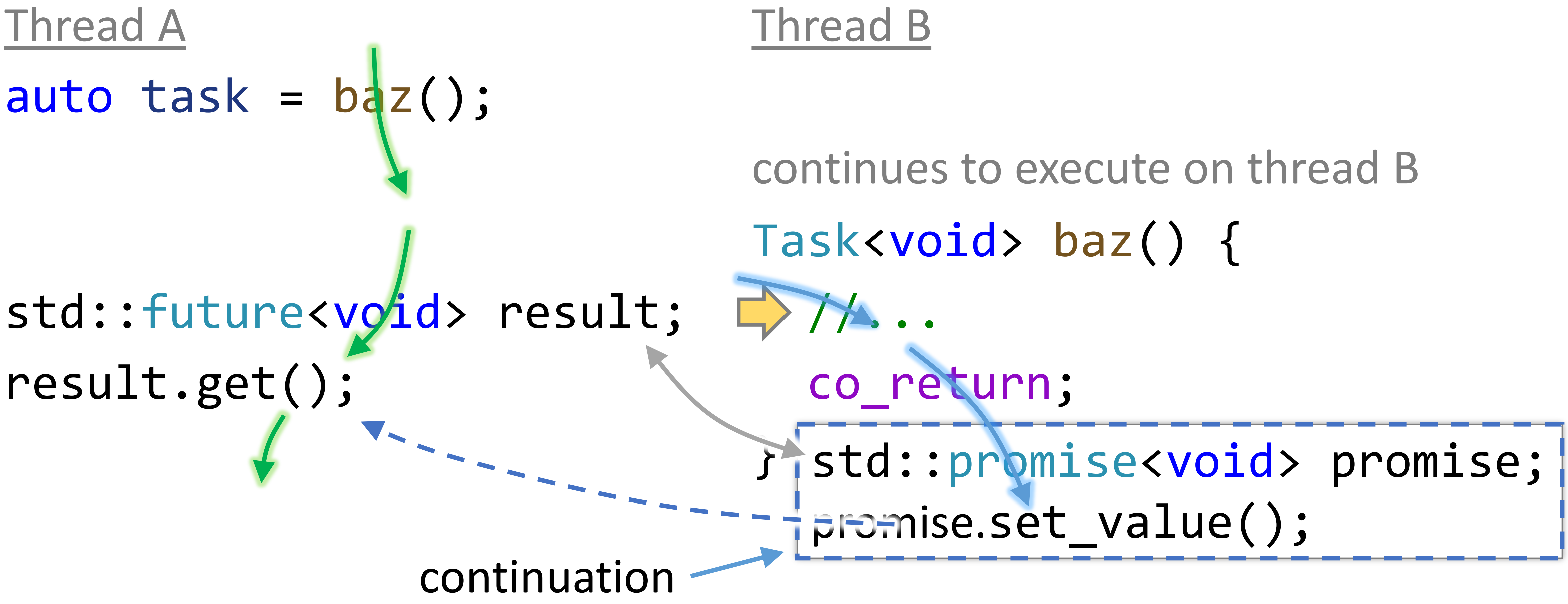
```
auto task = baz();
//...
// are we there yet?
auto result =
  getResult(task);
```

Thread B

continues to execute on thread B

```
Task<void> baz() {
⇨ //...
  co_return;
}
```

# Getting result from task



Thread A

```
auto task = baz();


std::future<void> result;
result.get();
```

Thread B

continues to execute on thread B

```
Task<void> baz() {
  //...
  co_return;
}
```

```
std::promise<void> promise;
promise.set_value();
```

continuation

# Getting result from task

```
template<typename T>
SyncWaitImpl<ResultOfAwait<T&&>> syncWaitImpl(T &&task) {
  co_return co_await std::forward<T>(task);
}

template<typename T>
auto syncWait(T &&task) {
  return syncWaitImpl(std::forward<T>(task))
    .result.get();
}
```

# Getting result from task

```
template<typename T>
struct SyncWaitImpl {
  struct promise_ty
    //...
  };

  std::future<T> result;
};
```

```
template<typename T>
auto syncWait(T &&task) {
  return syncWaitImpl(std::forward<T>(task))
    .result.get();
}
```

# Getting result from task

```
struct promise_type {
  SyncWaitImpl get_return_object() {
    return { promise.get_future() };
  }
  std::suspend_never initial_suspend() noexcept { return {}; }
  std::suspend_never final_suspend() noexcept { return {}; }
  void return_value(T &&value) {
    promise.set_value(std::move(value));
  }
  void unhandled_exception() {
    promise.set_exception(std::current_exception());
  }

  std::promise<T> promise;
};
```

# Getting result from task

```
auto task = bar();
auto result = syncWait(task);
```

# Getting result from task

```
Task<int> foo() {
  std::cout << "foo(): about to return\n";
  co_return 42;
}
Task<int> bar() {
  const auto result = foo();
  std::cout << "bar(): about to co_await\n";
  const int i = co_await result;
  std::cout << "bar(): about to return\n";
  co_return i + 23;
}

auto result = syncWait(bar());
```

# Making awaiting thread-safeish

`Task<T> Promise<T>`

# Making awaiting thread-safeish

```
template<typename T>
struct Promise {
  //...
  auto final_suspend() noexcept {
    struct FinalAwaitable { /*...*/ };
    return FinalAwaitable{};
  }
  //...
  enum class State {
    Started,
    AttachedContinuation,
    Finished
  };
  std::atomic<State> state = { State::Started };
};
```

# Making awaiting thread-safeish

```
template<typename T>
struct Promise {
  //...
  auto final_suspend() noexcept {
    struct FinalAwaitable { /*...*/ };
```

```
struct FinalAwaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(std::coroutine_handle<Promise<T>> thisCoro) {
    auto &promise = thisCoro.promise();
    const auto oldState = promise.state.exchange(State::Finished);
    if (oldState == State::AttachedContinuation)
      promise.continuation();
  }
  void await_resume() const noexcept {}
};
```

# Making awaiting thread-safeish

```
template<typename T>
struct Promise {
  //...
  //...
  bool isReady() const noexcept;
  //...
  std::variant<std::monostate, T, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
  enum class State { Started, AttachedContinuation, Finished };
  std::atomic<State> state = { State::Started };
};
```

```
bool isReady() const noexcept {
  // return result.index() != 0;
  return state == State::Finished;
}
```

# Making awaiting thread-safeish

```
template<typename T>
struct [[nodiscard]] Task {
  //...
      promise.continuation = continuation;
      auto expectedState = State::Started;
      return promise.state
        .compare_exchange_strong(expectedState,
                                 State::AttachedContinuation);
  }
  //...
};
```

If `state` was `Started`
compare-exchange succeeds
returning `true` → coroutine is suspended

If `state` was `Finished`
compare-exchange fails
returning `false` → coroutine is not suspended

# Iteration 2

```
Task<size_t
  std::cout                                      file async\n";
  const aut                                      'main.cpp" };
  std::cout                                      n (size "
    << resul
  co_return
}

int main()
  auto task
  std::cout
}
```

```
                                                 ead file async
                                                 ; file
                                                 ; file
                                                 ng coro
                                                 eturn (size 120)
                                                 ead: exiting
 result: 120
Promise: dtor
```

# Drawbacks of eager tasks

Thread A

```
void qux() {
   auto task = readFile();
   throw "oops...";
   task.~Task() ;
}
```

Thread B

continues to execute on thread B

```
auto work = [this, coro]() {
   //...
⇨  coro();
   //...
};
```

# State of the art solution so far: lazy tasks

Use `cppcoro` by Lewis Baker

https://github.com/lewissbaker/cppcoro

# State of the art solution so far: lazy tasks

code from Iteration 1

```
template<typename T>
struct Promise {
  Task<T> get_return_object() noexcept { return { this }; }
  std::suspend_always initial_suspend() noexcept { return {}; }
  auto final_suspend() noexcept;
  template<typename U>
  void return_value(U &&value)
    noexcept(std::is_nothrow_constructible_v<T, decltype(std::forward<U>(value))>);
  void unhandled_exception()
    noexcept(std::is_nothrow_constructible_v<std::exception_ptr, std::exception_ptr>);
  bool isReady() const noexcept;
  T &&getResult();

  std::variant<std::monostate, T, std::exception_ptr> result;
  std::coroutine_handle<> continuation;
};
```

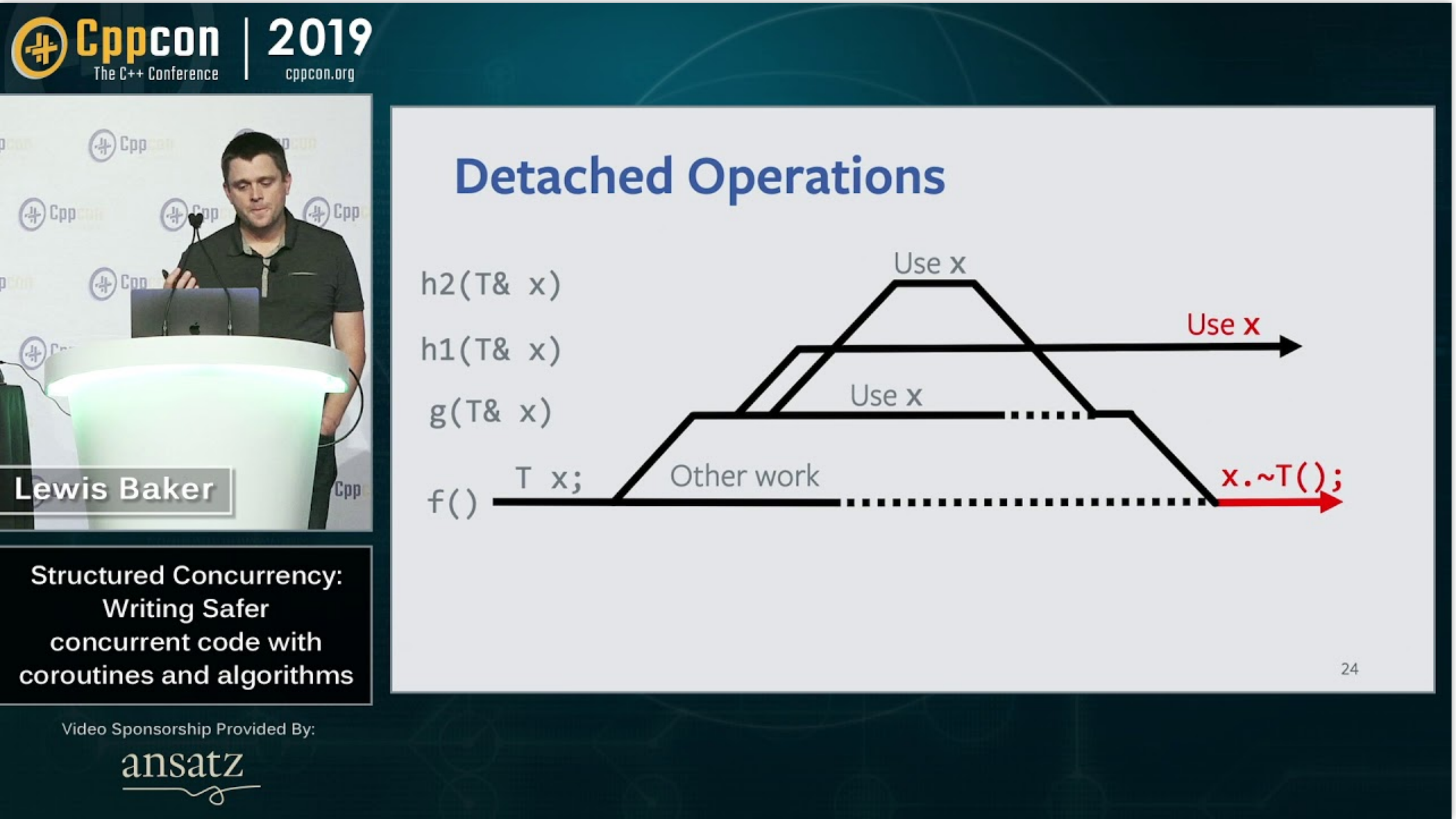
# State of the art solution so far: lazy tasks

```
void qux() {
  auto task = readFile(); // does not start yet
  throw "oops..."; // safe to cleanup
  syncWait(task); // awaiting starts the operation
}
```

# State of the art solution so far: lazy tasks

Use cppcoro by Lewis Baker

https://github.com/lewissbaker/cppcoro



CppCon The C++ Conference | 2019 cppcon.org

Lewis Baker

Structured Concurrency: Writing Safer concurrent code with coroutines and algorithms

Video Sponsorship Provided By: ansatz

## Detached Operations

https://youtu.be/1Wy5sq3s2rg

# Thanks for coming!

# Understanding C++ coroutines by example

Pavel Novikov

🐦 @cpp_ape

R&D Align Technology

align

Thanks to Lewis Baker for feedback!

I owe you beer 🍻

Slides: [https://git.io/JJvLX](https://git.io/JJvLX)

# Bonus slides

# getAwaitable()

```
template<typename T>
auto getAwaitableImpl(T &&a, int) ->
  decltype(std::forward<T>(a).operator co_await()) {
  return std::forward<T>(a).operator co_await();
}
template<typename T>
auto getAwaitableImpl(T &&a, long) ->
  decltype(operator co_await(std::forward<T>(a))) {
  return operator co_await(std::forward<T>(a));
}
template<typename T, typename U>
T &&getAwaitableImpl(T &&a, U) {
  return static_cast<T&&>(a);
}
```

```
template<typename T>
auto getAwaitable(T &&a) {
  return getAwaitableImpl(a, 42);
}
```

# ResultOfAwait<T>

```
template<typename T>
using ResultOfAwait =
  std::decay_t<decltype(
    getAwaitable(std::declval<T>()).await_resume()
  )>;
```

# tid

```
struct TidMark {
  friend
  std::ostream &operator<<(std::ostream &s, TidMark) {
    s << "(tid=" << std::this_thread::get_id() << ')';
    return s;
  }
} const tid;

std::cout << tid;
```

# State machine using coroutines

Events:

```
struct Open {};
struct Close {};
struct Knock {};
```

```
enum class State {
  Closed,
  Open
};

struct Door {
  State state = State::Closed;
  template<typename E>
  void onEvent(E);
};
```

# State machine using ~~coroutines~~ switch

```
void onEvent(E) {
 switch (state) {
 case State::Closed:
  if constexpr (isSame<E, Open>) {
   state = State::Open;
  }
  else if constexpr (isSame<E, Knock>) {
   shout("Come in, it's open!"); // no transition
  }
  break;
 case State::Open:
  if constexpr (isSame<E, Close>)
   state = State::Closed;
 }
}
```

# State machine using ~~coroutines~~ switch

```
Door door;
door.onEvent(Open{});  // Closed -> Open
door.onEvent(Close{}); // Open -> Closed
door.onEvent(Knock{});
door.onEvent(Close{}); // Closed -> Closed
```

output:

```
Come in, it's open!
```

# State machine using coroutines

```
StateMachine getDoor() {
  for (;;) {
    //closed
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      // open
      co_await Event<Close>{};
    }
  }
}
```

# State machine using coroutines

```
StateMachine getDoor() {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout("Come in, it's open!");
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

# State machine using coroutines

```
template<typename... Events>
struct Event {};

struct StateMachine {
  struct promise_type;

  template<typename E>
  void onEvent(E e);

  ~StateMachine() { coro.destroy(); }
  StateMachine(const StateMachine &) = delete;
  StateMachine &operator=(const StateMachine &) = delete;

private:
  StateMachine(std::coroutine_handle<promise_type> coro) : coro{ coro } {}
  std::coroutine_handle<promise_type> coro;
};
```

# State machine using coroutines

```
struct promise_type {
  using CoroHandle = std::coroutine_handle<promise_type>;
  StateMachine get_return_object() noexcept {
    return { CoroHandle::from_promise(*this) };
  }
  std::suspend_never initial_suspend() const noexcept { return {}; }
  std::suspend_always final_suspend() const noexcept { return {}; }
  template<typename... E>
  auto await_transform(Event<E...>) noexcept;
  void return_void() noexcept {}
  void unhandled_exception() noexcept {}

  std::any currentEvent;
  bool (*isWantedEvent)(const std::type_info&) = nullptr;
};
```

# StateMachine::promise_type

```cpp
template<typename... E>
auto await_transform(Event<E...>) noexcept {
  isWantedEvent = [](const std::type_info &type)->bool {
    return ((type == typeid(E)) || ...);
  };

  struct Awaitable { /*...*/ };
  return Awaitable{ &currentEvent };
}
```

# StateMachine::promise_type

```
struct Awaitable {
  bool await_ready() const noexcept { return false; }
  void await_suspend(CoroHandle) noexcept {}
  std::variant<E...> await_resume() const {
    std::variant<E...> event;
    (void)((
      currentEvent->type() == typeid(E) ?
        (event = std::move(*std::any_cast<E>(currentEvent)), true) :
        false
    ) || ...);
    return event;
  }
  const std::any *currentEvent;
};
```

# State machine using coroutines

```
struct StateMachine {
  //...
  template<typename E>
  void onEvent(E &&e) {
    auto &promise = coro.promise();
    if (promise.isWantedEvent(typeid(E))) {
      promise.currentEvent = std::forward<E>(e);
      coro();
    }
  }
  //...
};
```

# State machine using coroutines

```
auto door = getDoor();
door.onEvent(Open{});  // Closed -> Open
door.onEvent(Close{}); // Open -> Closed
door.onEvent(Knock{});
door.onEvent(Close{}); // Closed -> Closed
```

output:

```
Come in, it's open!
```

# State machine using coroutines

```
StateMachine getDoor(std::string answer) {
closed:
  for (;;) {
    auto e = co_await Event<Open, Knock>{};
    if (std::holds_alternative<Knock>(e)) {
      shout(answer);
    }
    else if (std::holds_alternative<Open>(e)) {
      goto open;
    }
  }
open:
  co_await Event<Close>{};
  goto closed;
}
```

# State machine using coroutines

```
auto door = getDoor("Occupied!");
door.onEvent(Open{});  // Closed -> Open
door.onEvent(Close{}); // Open -> Closed
door.onEvent(Knock{});
door.onEvent(Close{}); // Closed -> Closed
```

output:

```
Occupied!
```