

Asynchronous C++ programming

Pavel Novikov

 @cpp_ape

Align Technology R&D

align

What's ahead

- `std::async`, `std::future`,
`std::promise`,
`std::packaged_task`
- PPL: `concurrency::task`,
continuations, cancellation,
task composition

What's ahead

- `std::async`, `std::future`,
`std::promise`,
`std::packaged_task`
- PPL: `concurrency::task`,
continuations, cancellation,
task composition
- coroutines:
`co_await`, `co_return`,
generators,
implementation details

What's ahead

- `std::async`, `std::future`,
`std::promise`,
`std::packaged_task`
- PPL: `concurrency::task`,
continuations, cancellation,
task composition
- coroutines:
`co_await`, `co_return`,
generators,
implementation details
- future?

Metaphor

Неудачная метафора подобна котёнку с дверцей

Internets

Ineffectual metaphor is like a kitten with a hatch

literal translation from Russian

Metaphor

Synchronous variant:

```
boilWater();
```

```
makeTea();
```

```
drinkTea();
```

Metaphor

Asynchronous variant:

```
kettle.boilWaterAsync();  
playVideogamesFor(5min);  
kettle.waitWaterToBoil();  
makeTeaAsync();  
watchYouTubeFor(2min);  
drinkTea();
```

std::async



```
std::future<void> tea =  
    std::async(std::launch::async, []{  
        boilWater();  
        makeTea();  
    });  
watchPeoplePlayGamesOnYouTubeFor(7min);  
tea.get();  
drinkTea();
```


std::async



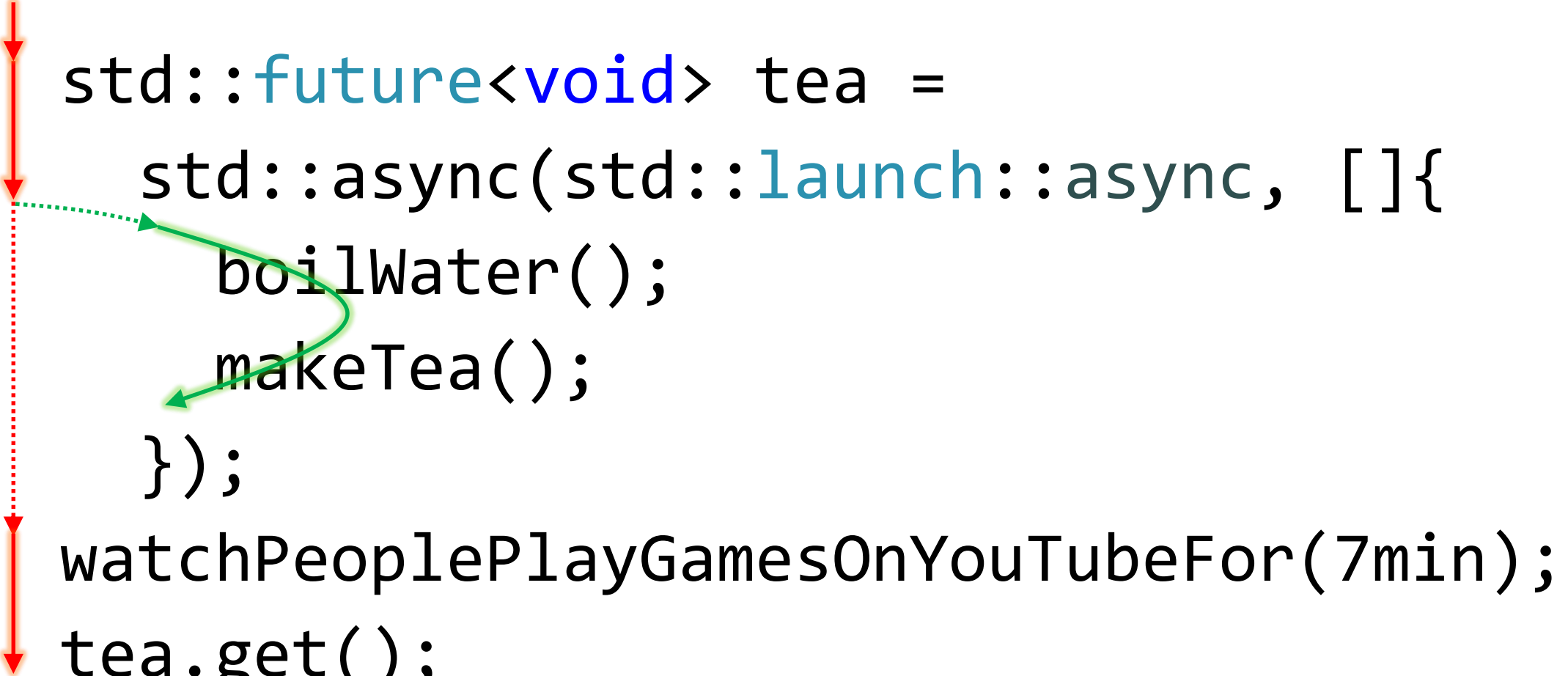
```
std::future<void> tea =  
    std::async(std::launch::async, []{  
        boilWater();  
        makeTea();  
    });  
watchPeoplePlayGamesOnYouTubeFor(7min);  
tea.get();  
drinkTea();
```

std::async

```
std::future<void> tea =  
    std::async(std::launch::async, []{  
        boilWater();  
        makeTea();  
    });  
watchPeoplePlayGamesOnYouTubeFor(7min);  
tea.get();  
drinkTea();
```

std::async

```
std::future<void> tea =  
    std::async(std::launch::async, []{  
        boilWater();  
        makeTea();  
    });  
watchPeoplePlayGamesOnYouTubeFor(7min);  
tea.get();  
drinkTea();
```



std::async

```
std::future<void> tea =  
    std::async(std::launch::async, []{  
        boilWater();  
        makeTea();  
    });  
watchPeoplePlayGamesOnYouTubeFor(7min);  
tea.get();  
drinkTea();
```

std::async


```
std::future<void> tea =  
    std::async(std::launch::async, []{  
        boilWater();  
        makeTea();  
    });  
watchPeoplePlayGamesOnYouTubeFor(7min);  
tea.get();  
drinkTea();
```

std::packaged_task



```
std::packaged_task<void()> task{ []{  
    boilWater();  
    makeTea();  
} };  
std::future<void> tea = task.get_future();  
help.execute(std::move(task));  
tea.get();  
drinkTea();
```

std::packaged_task



```
std::packaged_task<void()> task{ []{  
    boilWater();  
    makeTea();  
} };  
std::future<void> tea = task.get_future();  
help.execute(std::move(task));  
tea.get();  
drinkTea();
```

std::packaged_task

```
std::packaged_task<void()> task{ []{
```

```
    boilWater();
```

```
    makeTea();
```

```
};
```

```
std::future<void> tea = task.get_future();
```

```
help.execute(std::move(task));
```

```
tea.get();
```

```
drinkTea();
```


std::packaged_task

```
std::packaged_task<void()> task{ []{
```

```
    boilWater();
```

```
    makeTea();
```

```
};
```

```
std::future<void> tea = task.get_future();
```

```
help.execute(std::move(task));
```

```
tea.get();
```

```
drinkTea();
```

std::packaged_task

```
std::packaged_task<void()> task{ []{  
    boilWater();  
    makeTea();  
} };  
std::future<void> tea = task.get_future();  
help.execute(std::move(task));  
tea.get();  
drinkTea();
```

std::packaged_task

std::packaged_task<void()> task{ []{

boilWater();

makeTea();

};

std::future<void> tea = task.get_future();

help.execute(std::move(task));

tea.get();

drinkTea();

std::promise

```
std::promise<int> promise;
std::future<int> tea = promise.get_future();
auto task = [p = std::move(promise)]() mutable {
    try {
        boilWater();
        makeTea();
        p.set_value(42);
    }
    catch (...) { p.set_exception(std::current_exception()); }
};
help.execute(std::move(task));
tea.get();
```

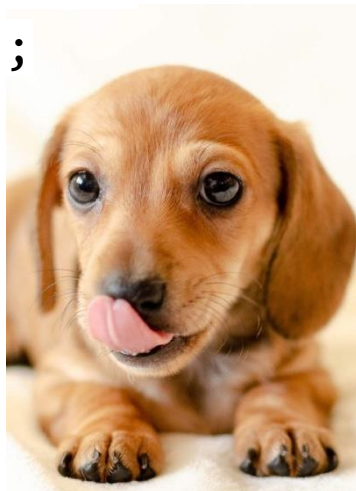
std::promise

```
std::promise<int> promise;
std::future<int> tea = promise.get_future();
auto task = [p = std::move(promise)]() mutable {
    try {
        boilWater();
        makeTea();
        p.set_value(42);
    }
    catch (...) { p.set_exception(std::current_exception()); }
};
help.execute(std::move(task));
tea.get();
```

The diagram illustrates the relationship between `std::promise` and `std::future`. Two boxes on the right, labeled `std::promise` and `std::future`, have arrows pointing to a central box labeled `shared state`. This indicates that both `std::promise` and `std::future` objects share the same underlying state.

Idiom: asynchronous value update

```
struct Widget {  
    std::future<void> updateValue();  
    std::string getValue() const;  
  
private:  
    struct State {  
        std::mutex mutex;  
        std::string value;  
    };  
    std::shared_ptr<State> m_state =  
        std::make_shared<State>();  
};
```



```
std::future<void> Widget::updateValue() {  
    return std::async(  
        [statePtr = std::weak_ptr{m_state}] {  
            auto newValue = getUpdatedValue();  
            if (auto state = statePtr.lock()) {  
                std::lock_guard lock(state->mutex);  
                state->value = std::move(newValue);  
            }  
        });  
}  
  
std::string Widget::getValue() const {  
    std::lock_guard lock(m_state->mutex);  
    return m_state->value;  
}
```

Parallel Patterns Library

- released by Microsoft together with Visual Studio 2010 (+ lamdas)
- a subset (PPLX) is implemented in a cross-platform library C++ REST SDK
<https://github.com/Microsoft/cpprestsdk>

PPL concurrency::task 101

```
#include <ppltasks.h>  
using namespace concurrency;
```

```
task<T>
```

```
auto t = task<T>{f};  
auto t = create_task(f);
```

```
task_completion_event<T>
```

```
#include <future>
```

```
std::future<T>
```

```
auto t = std::async(  
    std::launch::async, f);
```

```
std::promise<T>
```


Task unwrapping

```
std::future<void> makeTeaAsync();
```

```
std::future<std::future<void>> tea =  
    std::async([]() -> std::future<void> {  
        boilWater();  
        return makeTeaAsync();  
    });  
tea.get().get();  
drinkTea();
```

Task unwrapping

```
task<void> makeTeaAsync();
```



```
auto tea = task<void>{[]() -> task<void> {  
    boilWater();  
    return makeTeaAsync();  
} };  
tea.wait();  
drinkTea();
```

Task unwrapping

```
task<void> makeTeaAsync();
```

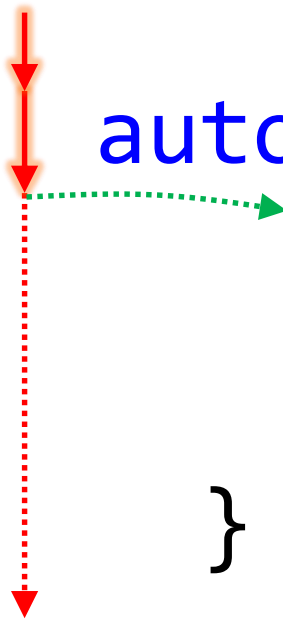


```
auto tea = task<void>{[]() -> task<void> {  
    boilWater();  
    return makeTeaAsync();  
} };  
tea.wait();  
drinkTea();
```

Task unwrapping

```
task<void> makeTeaAsync();
```

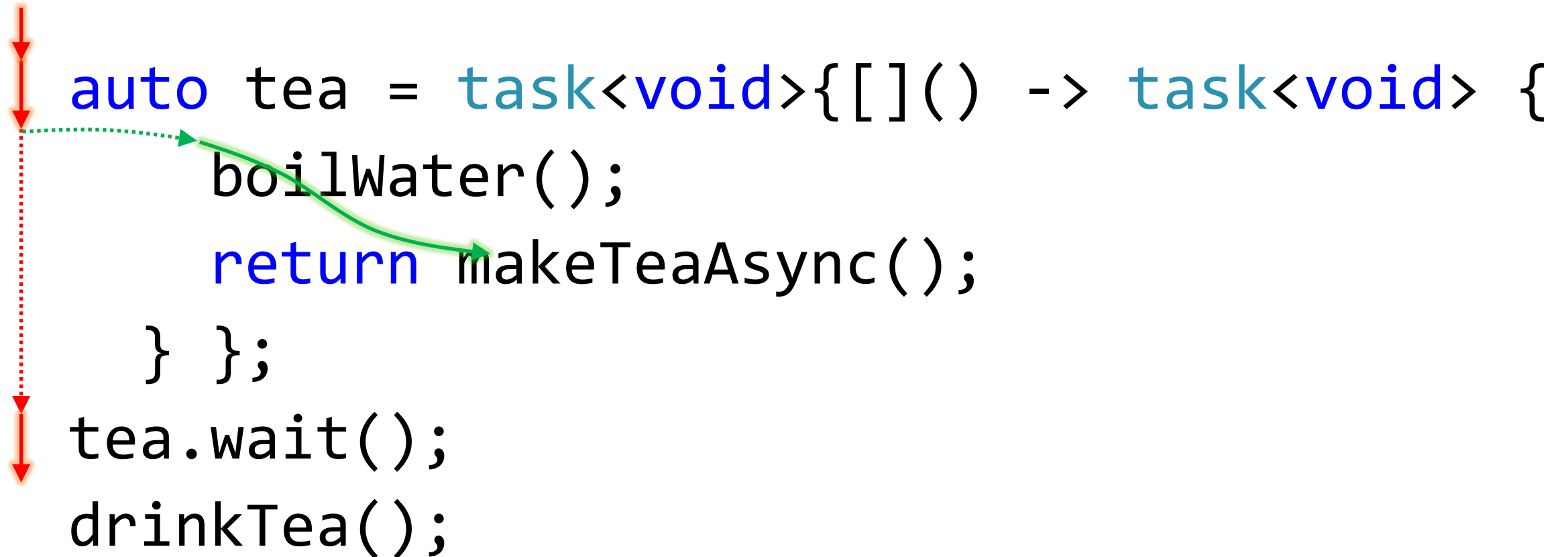
```
auto tea = task<void>{[]() -> task<void> {  
    boilWater();  
    return makeTeaAsync();  
} };  
tea.wait();  
drinkTea();
```



Task unwrapping

```
task<void> makeTeaAsync();
```

```
auto tea = task<void>{[]() -> task<void> {  
    boilWater();  
    return makeTeaAsync();  
} };  
tea.wait();  
drinkTea();
```



Task unwrapping

```
task<void> makeTeaAsync();
```

```
auto tea = task<void>{[]() -> task<void> {  
    boilWater();  
    return makeTeaAsync();  
} };  
tea.wait();  
drinkTea();
```

The diagram illustrates the process of task unwrapping. A red dashed arrow points from the lambda function's return type `task<void>` to the `return` statement. A green arrow points from the `return` statement to the `makeTeaAsync()` call. Another green arrow points from the `makeTeaAsync()` call to the lambda function's body. A red dashed arrow also points from the lambda function's return type `task<void>` to the `tea.wait()` call.

Task unwrapping

```
task<void> makeTeaAsync();
```

```
auto tea = task<void>{[]() -> task<void> {  
    boilWater();  
    return makeTeaAsync();  
} };  
tea.wait();  
drinkTea();
```

The diagram illustrates the process of task unwrapping. A red dashed arrow points from the lambda expression to the 'return' statement. A green arrow points from the 'return' statement to the 'makeTeaAsync()' call. Another green arrow points from the 'makeTeaAsync()' call to the lambda expression. A red dashed arrow points from the lambda expression to the 'tea.wait()' call.

Task unwrapping

```
task<void> makeTeaAsync();
```

```
auto tea = task<void>{[]() -> task<void> {  
    boilWater();  
    return makeTeaAsync();  
} };  
tea.wait();  
drinkTea();
```

The diagram illustrates the execution flow of the code. A red vertical arrow on the left indicates the main execution path. A blue dotted arrow points from the lambda function's return statement to the 'return' keyword. A green arrow points from the 'return' keyword to the 'makeTeaAsync()' call. A blue dotted arrow points from the 'makeTeaAsync()' call to the 'tea.wait()' call. A red vertical arrow also points to the 'tea.wait()' call.

Task unwrapping

```
task<void> makeTeaAsync();
```

```
auto tea = task<void>{[]() -> task<void> {  
    boilWater();  
    return makeTeaAsync();  
} };  
tea.wait();  
drinkTea();
```

Continuations

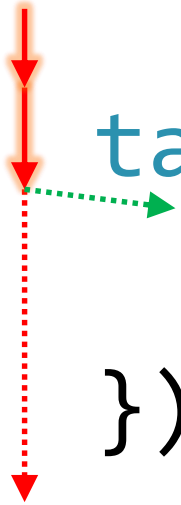


```
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

Continuations

```
↓  
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

Continuations

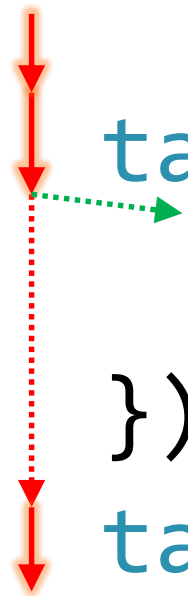


```
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

The diagram illustrates control flow with red arrows. A solid red arrow points down from the top left to the opening curly brace of the `water` task. A dotted red arrow continues down from the closing curly brace of the `water` task to the opening curly brace of the `tea` task. A green dotted arrow points from the opening curly brace of the `water` task to the `return` statement, indicating the return value being passed to the `then` method.

Continuations

```
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

A diagram illustrating control flow. A solid red arrow points down from the top left to the opening curly brace of the first lambda function. A green dotted arrow points from the opening curly brace of the first lambda function to the opening curly brace of the second lambda function. A red dotted arrow points down from the opening curly brace of the first lambda function to the opening curly brace of the second lambda function. A solid red arrow points down from the opening curly brace of the second lambda function to the end of the code block.

Continuations

```
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

The diagram illustrates the execution flow of the code. A vertical red arrow on the left indicates the main thread's execution path, starting from the top and moving down through the function definitions and the `tea.wait()` call. A green dotted arrow originates from the opening curly brace of the `water` task's lambda function and points to the `boilWater()` call. Another green dotted arrow originates from the opening curly brace of the `tea` task's lambda function and points to the `makeTea()` call. A red arrow also points to the opening curly brace of the `tea` task's lambda function, indicating the point where the main thread resumes execution after the `water` task has completed.

Continuations

```
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

The diagram shows a vertical red dashed line with downward-pointing arrows on the left side of the code, indicating the execution flow. A green dotted arrow points from the first lambda function's opening curly brace to the `boilWater()` call. Another green dotted arrow points from the second lambda function's opening curly brace to the `makeTea()` call.

Continuations

```
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

The diagram illustrates the flow of execution for the provided C++ code. A vertical red dashed line with downward-pointing arrows on the left side indicates the main thread's execution path. It starts at the top, moves down to the opening curly brace of the `water` task, then continues down past the closing curly brace of `water` to the opening curly brace of the `tea` task, and finally reaches the `tea.wait()` call. A green curved arrow originates from the `return` statement in the `water` task's lambda function and points back to the opening curly brace of the `tea` task's lambda function, representing the continuation of the `water` task's execution into the `tea` task.

Continuations

```
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

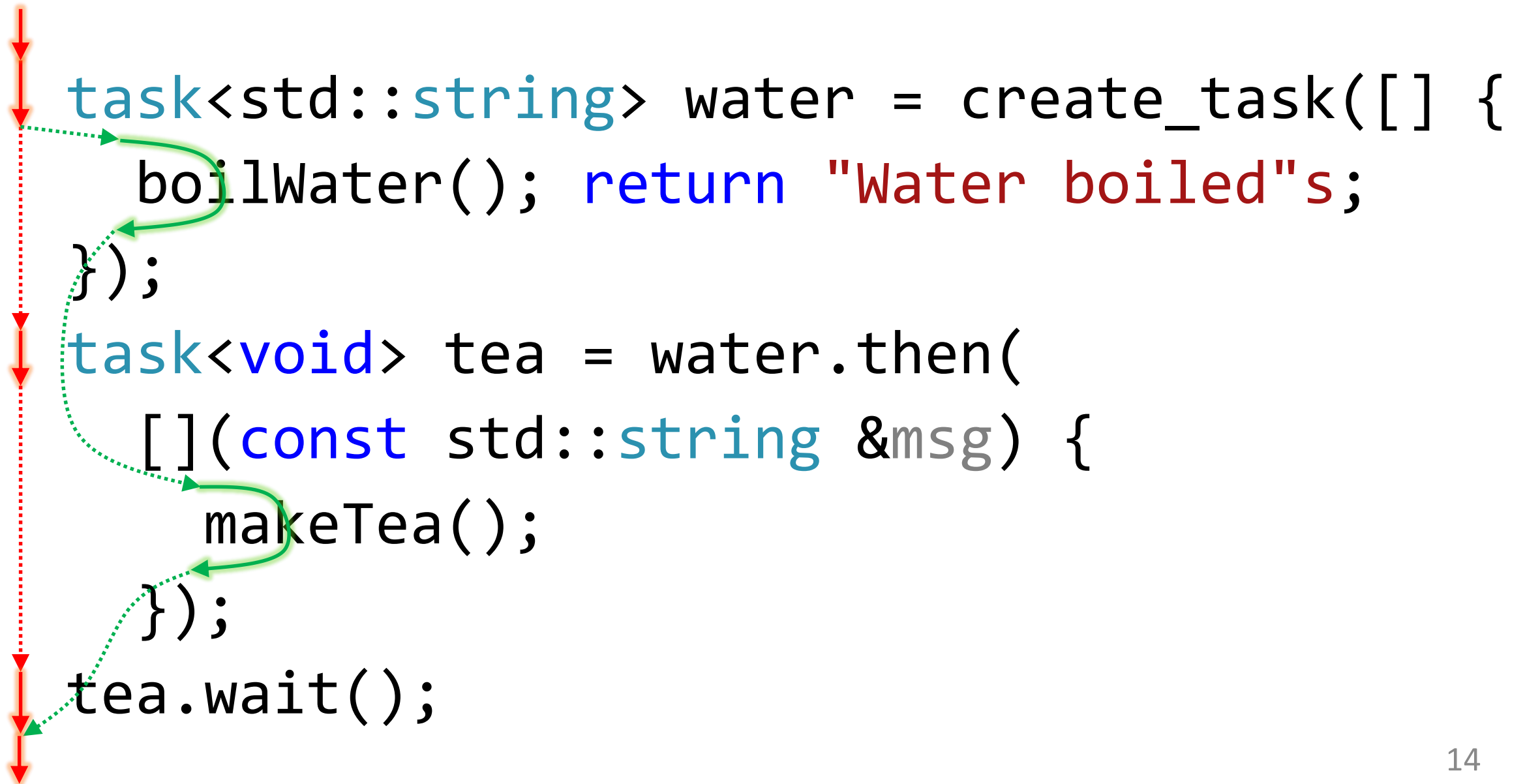
The diagram illustrates the execution flow of the provided C++ code. A vertical red dashed line with downward-pointing arrows indicates the main thread's execution path. It starts at the top, moves down to the opening curly brace of the `water` task, then to the opening curly brace of the `tea` task, and finally to the `tea.wait()` call. Green arrows show the continuation links: one from the `return` statement in the `water` task to the opening curly brace of the `tea` task, and another from the `makeTea()` call back to the opening curly brace of the `tea` task.

Continuations

```
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

The diagram illustrates the execution flow of the provided C++ code. A vertical red dashed line with downward-pointing arrows on the left side indicates the main thread's execution path. It starts at the top, moves down to the opening curly brace of the `water` task, then continues down past the closing curly brace of `water` to the opening curly brace of the `tea` task, and finally reaches the `tea.wait()` statement. Green solid arrows with arrowheads show the continuation of execution from the `water` task's body back to the `water` task's closing brace, and then from the `tea` task's body back to the `tea` task's closing brace. Dotted green lines also connect the `water` task's closing brace to the `tea` task's opening brace, and the `tea` task's closing brace to the `tea.wait()` statement, showing the return path of control.

Continuations



Exception handling



```
auto tea = create_task([]() -> int {
    throw std::runtime_error{ "BANG!" }; })
    .then([](const task<int> &t) {
        t.get();
        boilWater(); return "Water boiled"s;
    })
    .then([](const std::string &msg) {
        makeTea(); });
tea.wait();
```

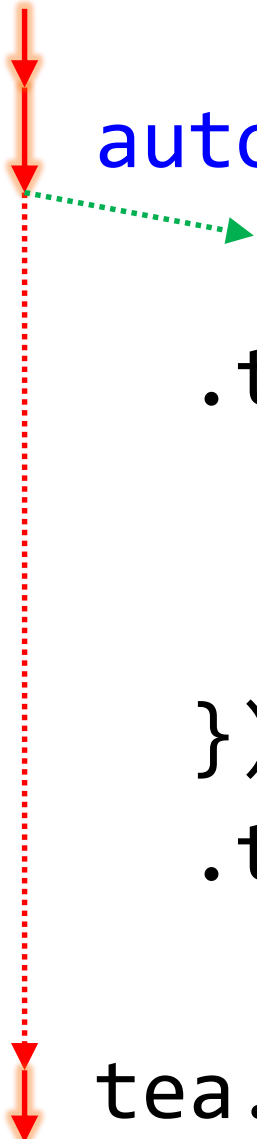
Exception handling



```
auto tea = create_task([]() -> int {
    throw std::runtime_error{ "BANG!" }; })
    .then([](const task<int> &t) {
        t.get();
        boilWater(); return "Water boiled"s;
    })
    .then([](const std::string &msg) {
        makeTea(); });
tea.wait();
```

Exception handling

```
auto tea = create_task([]() -> int {  
    throw std::runtime_error{ "BANG!" }; })  
    .then([](const task<int> &t) {  
        t.get();  
        boilWater(); return "Water boiled"s;  
    })  
    .then([](const std::string &msg) {  
        makeTea(); });  
tea.wait();
```



Exception handling

```
auto tea = create_task([]() -> int {  
    throw std::runtime_error{ "BANG!" }; })  
    .then([](const task<int> &t) {  
        t.get();  
        boilWater(); return "Water boiled"s;  
    })  
    .then([](const std::string &msg) {  
        makeTea(); });  
tea.wait();
```

Exception handling

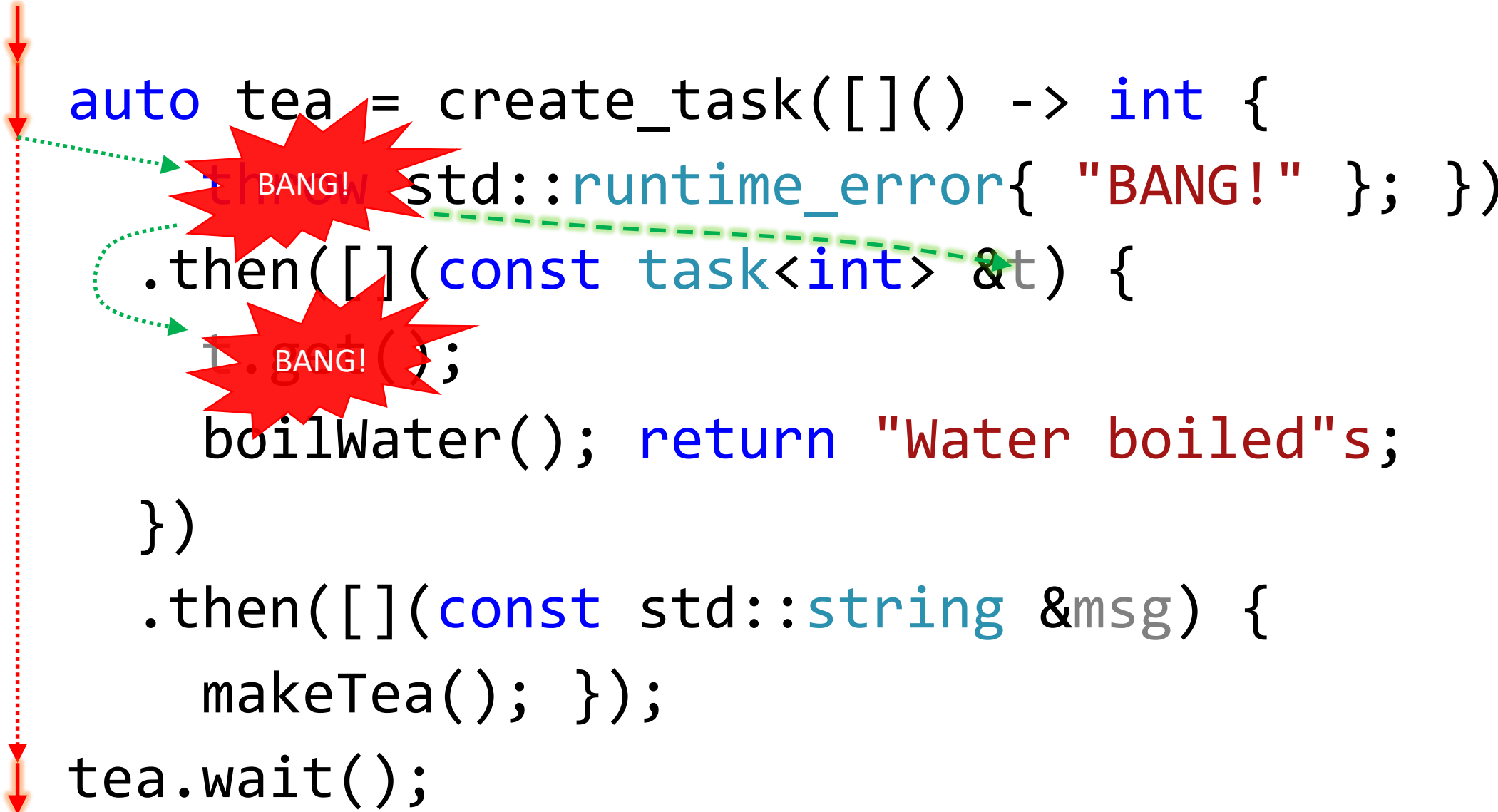
```
auto tea = create_task([]() -> int {
    throw std::runtime_error{ "BANG!" }; })
    .then([](const task<int> &t) {
        t.get();
        boilWater(); return "Water boiled"s;
    })
    .then([](const std::string &msg) {
        makeTea(); });
tea.wait();
```


Exception handling

```
auto tea = create_task([]() -> int {  
    throw std::runtime_error{ "BANG!" }; })  
    .then([](const task<int> &t) {  
        t.get();  
        boilWater(); return "Water boiled"s;  
    })  
    .then([](const std::string &msg) {  
        makeTea(); });  
tea.wait();
```

Exception handling

```
auto tea = create_task([]() -> int {
    throw std::runtime_error{ "BANG!" }; })
    .then([](const task<int> &t) {
        t.get();
        boilWater(); return "Water boiled"s;
    })
    .then([](const std::string &msg) {
        makeTea(); });
tea.wait();
```



Exception handling

```
auto tea = create_task([]() -> int {  
    throw std::runtime_error{ "BANG!" }; })  
    .then([](const task<int> &t) {  
        t.get();  
        boilWater(); return "Water boiled"s;  
    })  
    .then([](const std::string &msg) {  
        makeTea(); });  
tea.wait();
```

Exception handling

```
auto tea = create_task([]() -> int {  
    throw std::runtime_error{ "BANG!" }; })  
    .then([](const task<int> &t) {  
        t.get();  
        boilWater(); return "Water boiled"s;  
    })  
    .then([](const std::string &msg) {  
        makeTea(); });  
tea.wait();
```

Exception handling

```
auto tea = create_task([]() -> int {  
    throw std::runtime_error{ "BANG!" }; })  
    .then([](const task<int> &t) {  
        t.get();  
        boilWater(); return "Water boiled"s;  
    })  
    .then([](const std::string &msg) {  
        makeTea(); });  
tea.wait();
```

Cancellation

```
cancellation_token_source tokenSource;
```

```
create_task([token = tokenSource.get_token()] {  
    boilWater();  
    if (token.is_canceled())  
        cancel_current_task();//throws task_canceled{}  
    makeTea();  
})  
.wait();
```

Cancellation

```
cancellation_token_source tokenSource;
```

```
create_task([token = tokenSource.get_token()] {  
    boilWater();  
    if (token.is_canceled())  
        cancel_current_task();//throws task_canceled{}  
    makeTea();  
})  
.wait();
```

Usage:

```
tokenSource.cancel();
```

Cancellation callback

```
void boilWater(cancellation_token token)
{
    const auto registration =
        token.register_callback([] {
            stopBoiling();
        });
    boilWater();
    token.deregister_callback(registration);
}
```


Cancellation callback

```
void boilWater(cancellation_token token)
{
    const auto registration =
        token.register_callback([] {
            stopBoiling();
        });
    boilWater();
    token.deregister_callback(registration);
}
```

proposal [P0660](#) in
the C++ standard

Task composition: `when_all`

```
task<std::string> boilWaterAndMakeTeaAsync();
```

```
task<std::string> makeSandwichAsync();
```

```
task<std::string> tasks[] = {  
    boilWaterAndMakeTeaAsync(),  
    makeSandwichAsync()  
};
```

```
task<std::vector<std::string>> result =  
    when_all(std::begin(tasks), std::end(tasks));
```

Task composition: `when_all`

```
task<std::string> boilWaterAndMakeTeaAsync();
```

```
task<std::string> makeSandwichAsync();
```

```
task<std::string> tasks[] = {  
    boilWaterAndMakeTeaAsync(),  
    makeSandwichAsync()  
};
```

returns results of tasks



```
task<std::vector<std::string>> result =  
    when_all(std::begin(tasks), std::end(tasks));
```

Task composition: `when_any`

```
task<std::string> boilWaterAndMakeTeaAsync();
```

```
task<std::string> makeSandwichAsync();
```

```
task<std::string> tasks[] = {  
    boilWaterAndMakeTeaAsync(),  
    makeSandwichAsync()  
};
```

```
task<std::pair<std::string, size_t>> result =  
    when_any(std::begin(tasks), std::end(tasks));
```

Task composition: `when_any`

```
task<std::string> boilWaterAndMakeTeaAsync();
```

```
task<std::string> makeSandwichAsync();
```

```
task<std::string> tasks[] = {  
    boilWaterAndMakeTeaAsync(),  
    makeSandwichAsync()  
};
```

returns task result and its index



```
task<std::pair<std::string, size_t>> result =  
    when_any(std::begin(tasks), std::end(tasks));
```

Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```



Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```

a sea puppy




Idiom: do while

```
↓
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```

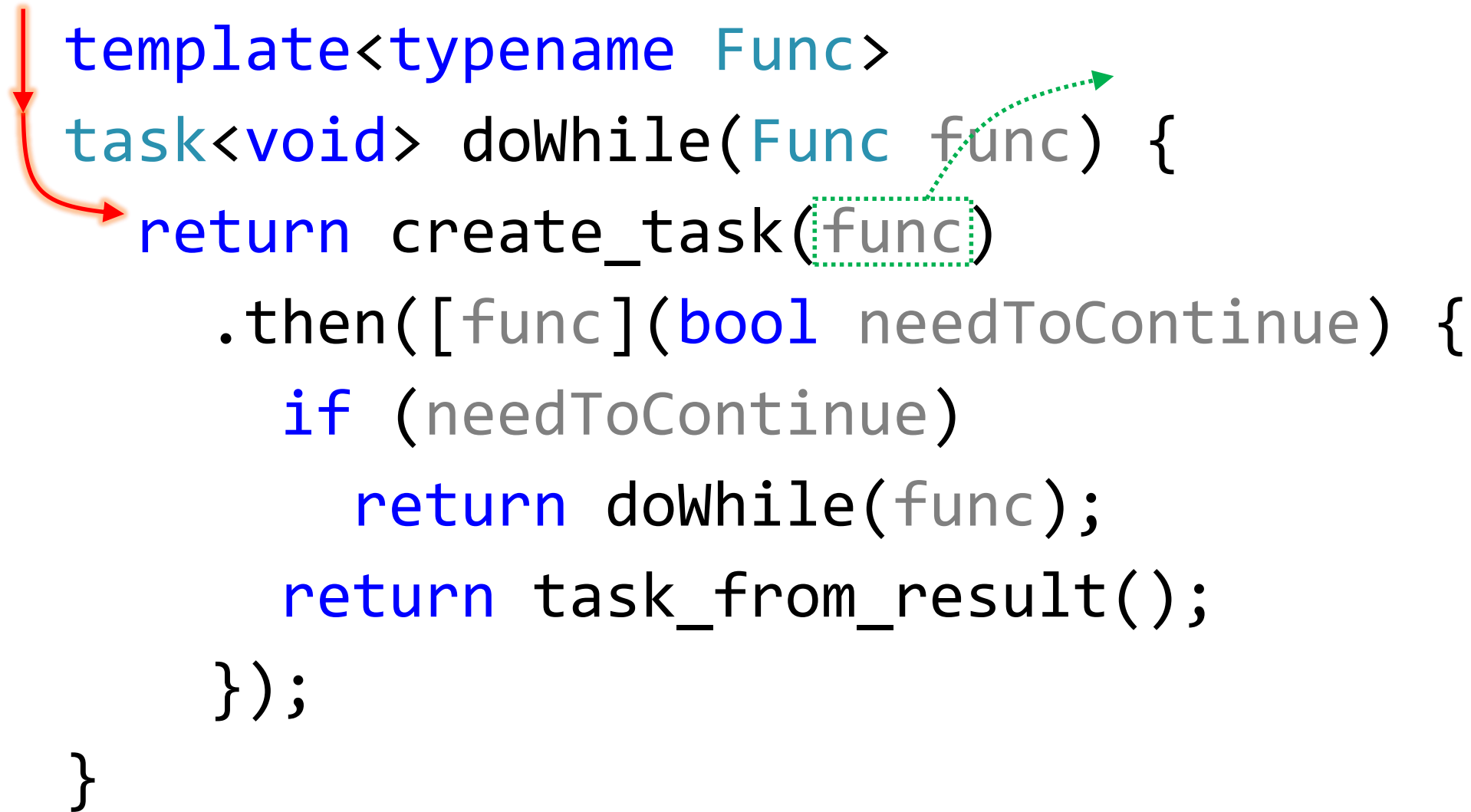

Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```



Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```

A diagram illustrating the 'do while' idiom. A red arrow points from the top left to the 'return' statement in the first line of the function body. Another red arrow points from the 'return' statement to the 'func' argument in the 'create_task' call. A green dashed box highlights the 'func' argument, and a green dashed arrow points from it to the 'func' parameter in the 'doWhile' function signature.

Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```

The diagram illustrates the flow of control in the provided C++ code. Red arrows indicate the execution path: starting from the beginning of the function, they point to the `return` statement, then to the `create_task` function call, and finally to the `.then` lambda function. A green dashed box highlights the `func` argument passed to `create_task`, with a green dashed arrow pointing from it to the `[func]` capture in the `.then` lambda, showing how the function object is passed to the continuation.

Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
        if (needToContinue)
            return doWhile(func);
        return task_from_result();
    });
}
```

The diagram illustrates the flow of control in the provided C++ code. Red arrows indicate the initial flow: one points to the function signature, another to the `return` statement, and a third to the `.then()` call. Green arrows and boxes highlight recursive logic: a green arrow points from the `func` parameter in `create_task(func)` to the `func` parameter in the `doWhile` call inside the `if` block; another green arrow points from the `needToContinue` parameter in the `if` block to the `needToContinue` parameter in the `then` call; and a green box encloses the `if` block and the `return task_from_result();` line, indicating the recursive loop.

Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```

The diagram illustrates the execution flow of the `doWhile` function. Red arrows show the initial call to `doWhile`, the return of `create_task(func)`, and the subsequent call to `.then()`. A green dotted arrow highlights the recursive call to `doWhile(func)` inside the `then` block, indicating the loop mechanism.

Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```

The diagram illustrates the flow of control in the provided C++ code. Red arrows show the execution path: starting from the function signature, moving to the `return` statement, then to the `then` block, and finally to the `if` block. A green dotted arrow highlights the recursive call path, starting from the `if` block, moving to the `return doWhile(func);` line, and then back to the `then` block, indicating a loop in the execution.

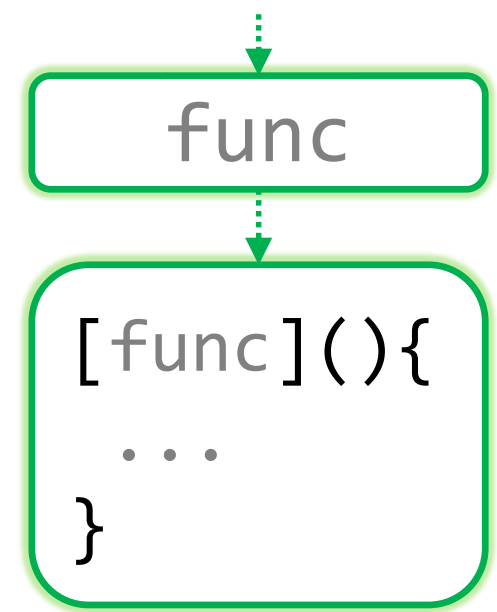
Idiom: do while

func

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```

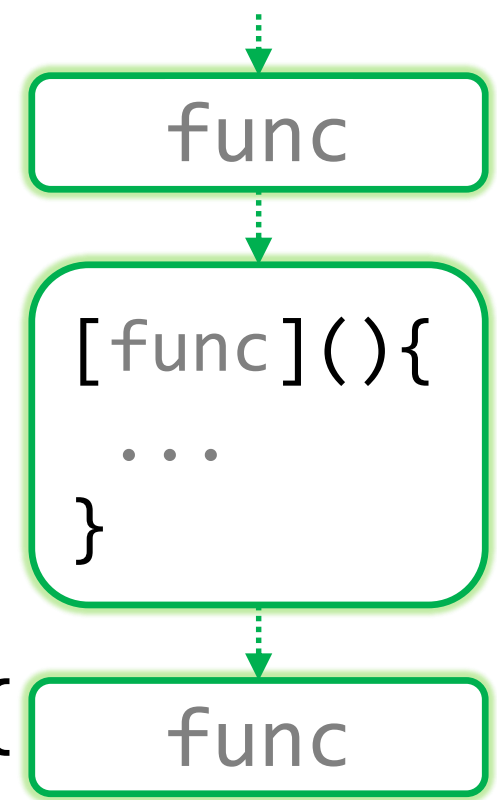
Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```



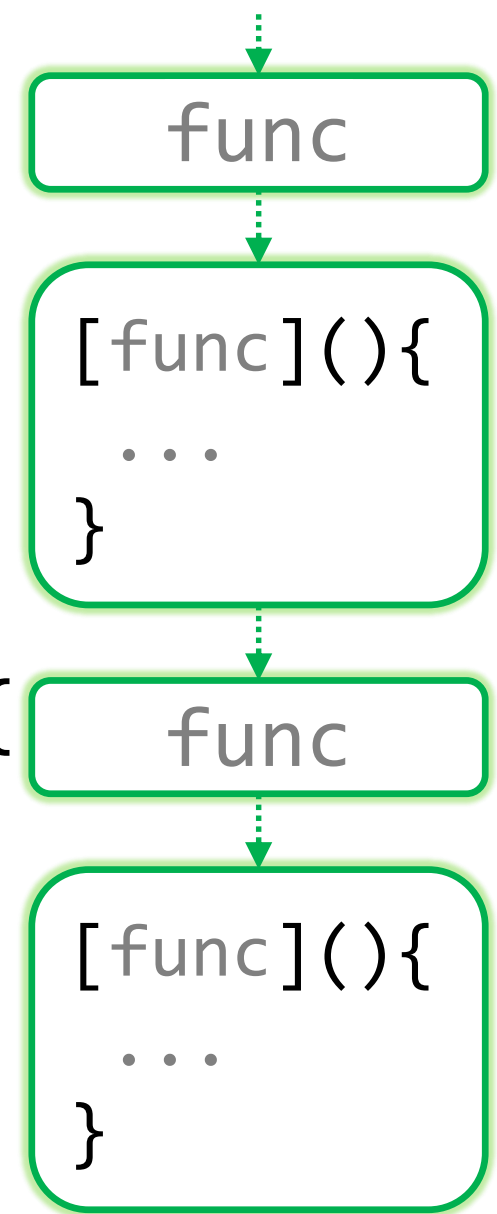
Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```



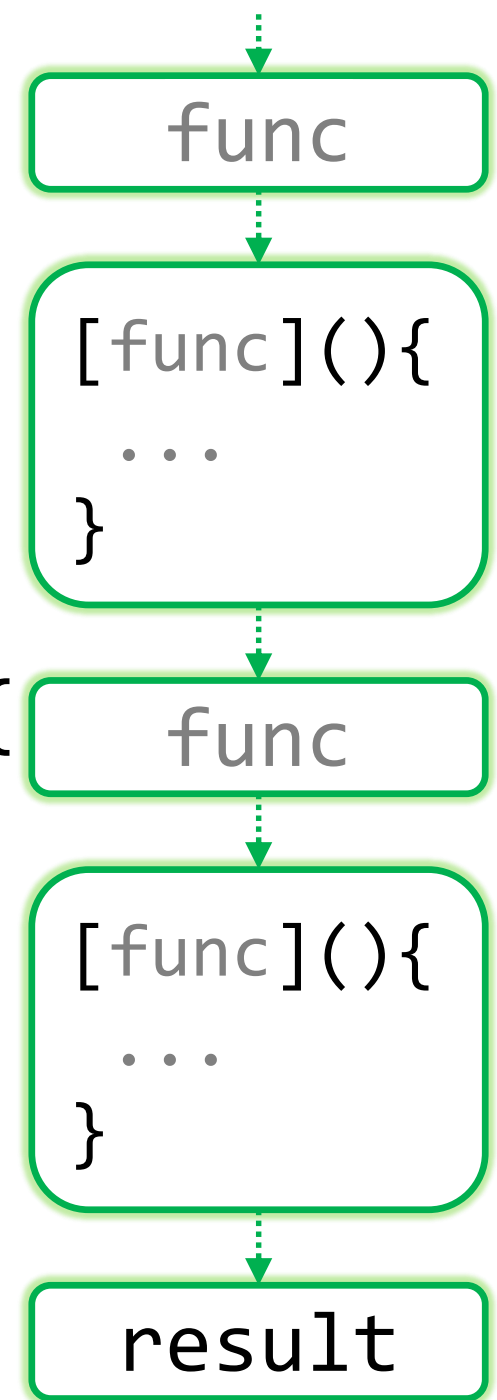
Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
        if (needToContinue)
            return doWhile(func);
        return task_from_result();
    });
}
```



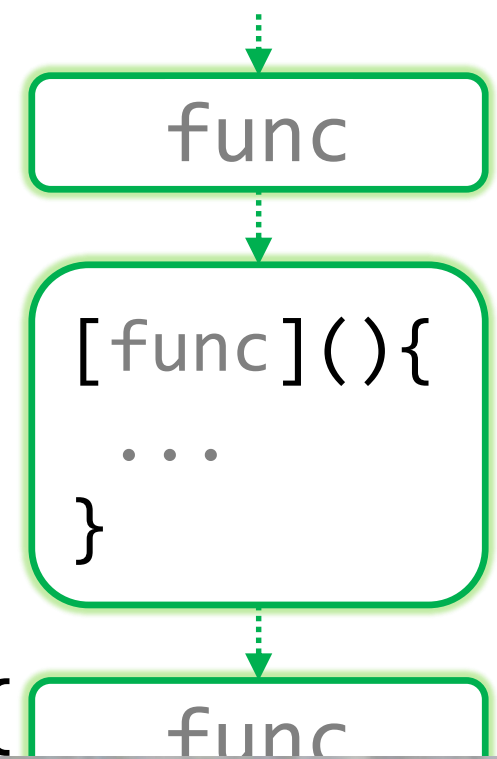
Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```



Idiom: do while

```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```



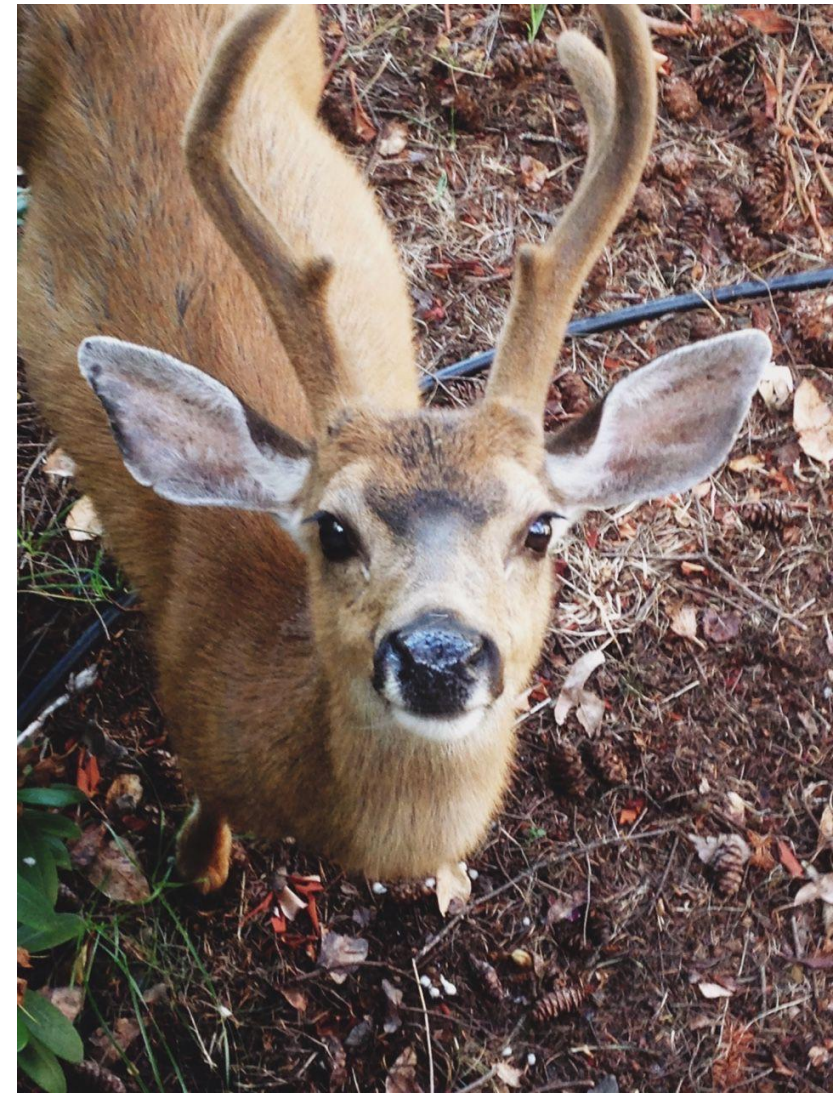
Idiom: cancellation of several requests

```
task<std::string> makeRequest(const std::string &request,
                             cancellation_token);

struct Gadget {
    task<std::string> makeRequest(const std::string &request) {
        return makeRequest(request, m_tokenSource.get_token());
    }
    void cancelAllRequests() {
        m_tokenSource.cancel();
        m_tokenSource = {};
    }
private:
    cancellation_token_source m_tokenSource;
};
```

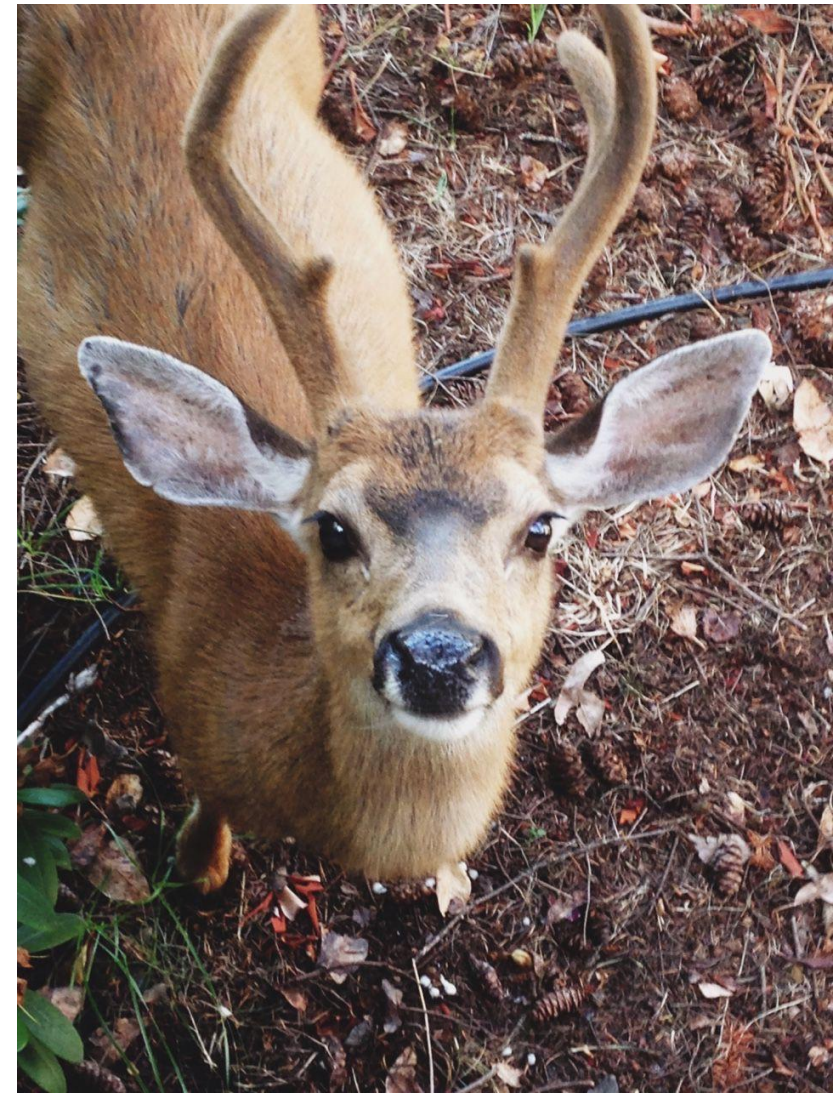
Idiom: continuation chaining

```
auto avatar = http::client::http_client{"https://reqres.in"}
    .request(http::methods::GET, "/api/users/1")
    .then([](const http::http_response &response) {
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get user");
        return response.extract_json();
    })
    .then([](const json::value &response) {
        const auto url = response.at("data").at("avatar").as_string();
        return http::client::http_client(url).request(http::methods::GET);
    })
    .then([](const concurrency::task<http::http_response> &result) {
        const auto response = result.get();
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get avatar");
        return response.extract_vector();
    });
```



Idiom: continuation chaining

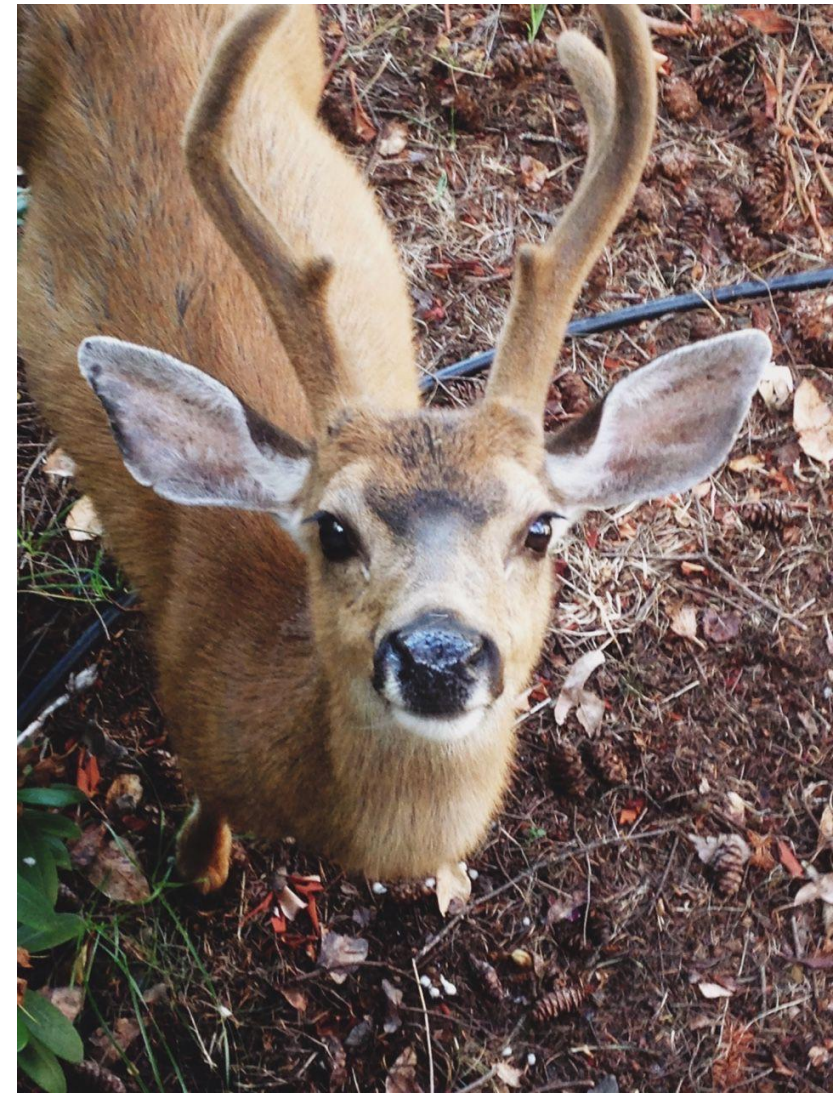
```
auto avatar = http::client::http_client{"https://reqres.in"}
    .request(http::methods::GET, "/api/users/1")
    .then([](const http::http_response &response) {
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get user");
        return response.extract_json();
    })
    .then([](const json::value &response) {
        const auto url = response.at("data").at("avatar").as_string();
        return http::client::http_client(url).request(http::methods::GET);
    })
    .then([](const concurrency::task<http::http_response> &result) {
        const auto response = result.get();
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get avatar");
        return response.extract_vector();
    });
```



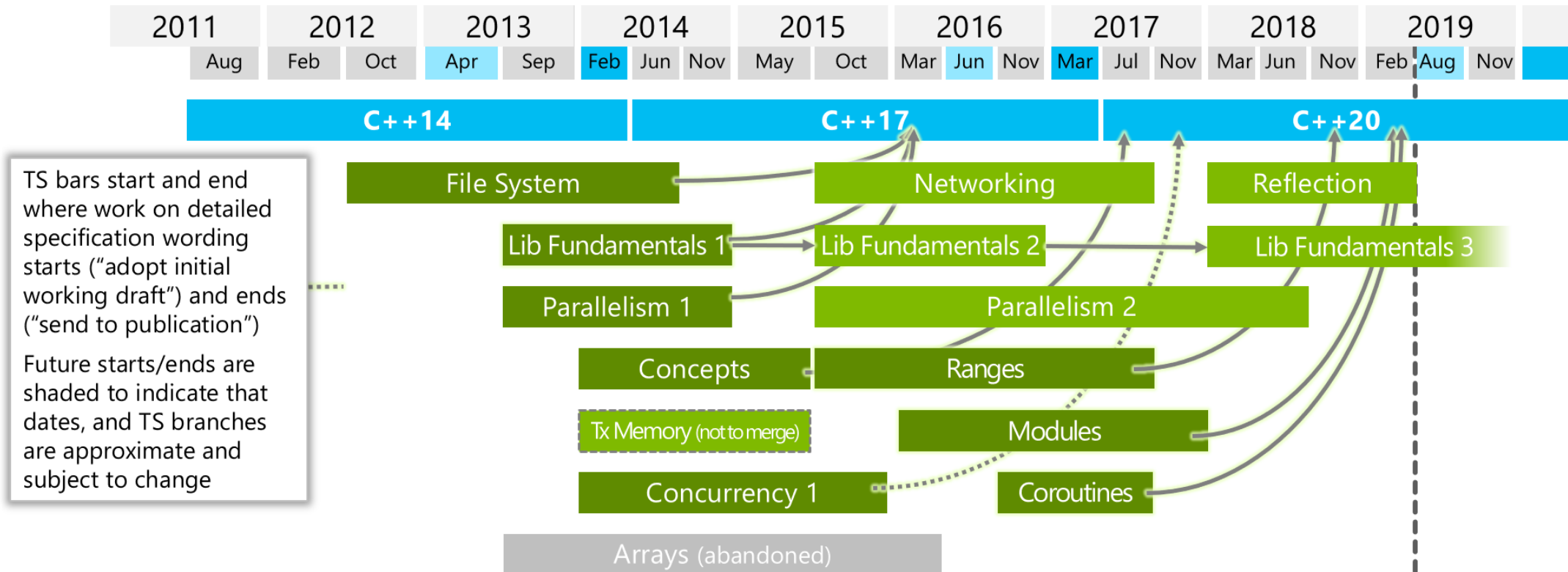
Idiom: continuation chaining

a forest puppy

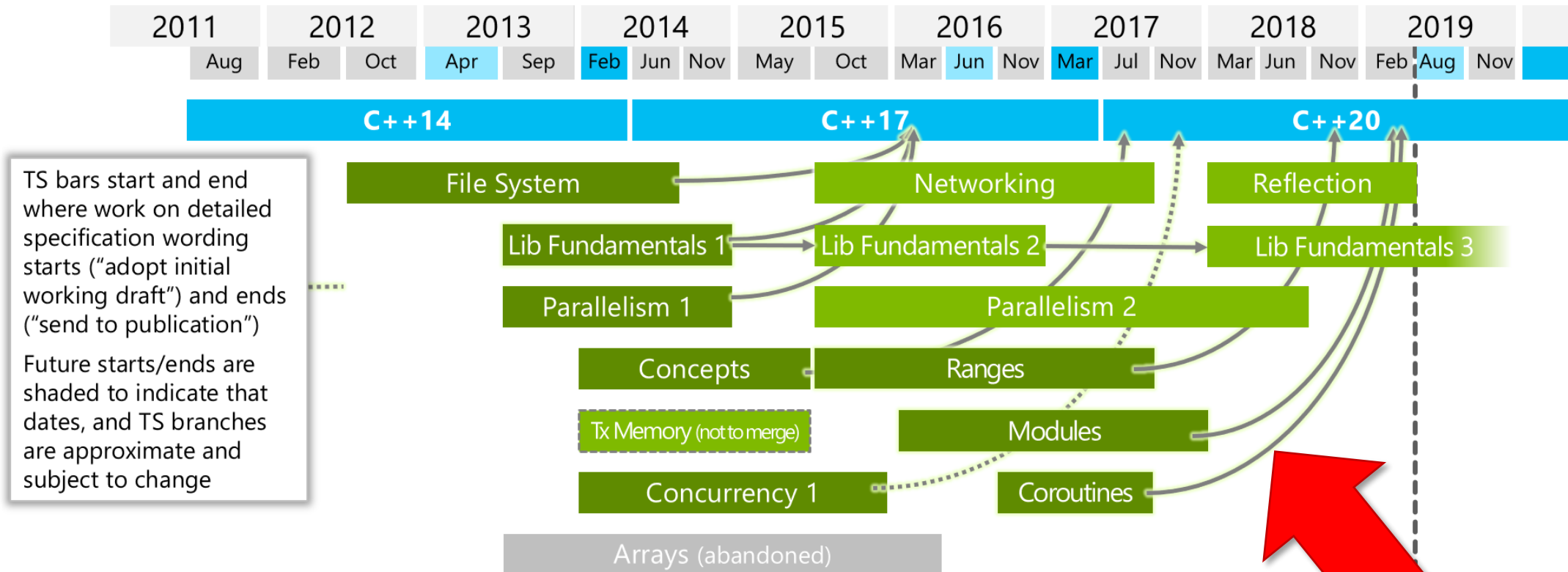
```
auto avatar = http::client::http_client{"https://reqres.in"}
    .request(http::methods::GET, "/api/users/1")
    .then([](const http::http_response &response) {
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get user");
        return response.extract_json();
    })
    .then([](const json::value &response) {
        const auto url = response.at("data").at("avatar").as_string();
        return http::client::http_client(url).request(http::methods::GET);
    })
    .then([](const concurrency::task<http::http_response> &result) {
        const auto response = result.get();
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get avatar");
        return response.extract_vector();
    });
```



<https://isocpp.org/std/status>



<https://isocpp.org/std/status>



TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send to publication")

Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change

Why do I need coroutines?

```
task<void> boilWaterAsync();
```

```
task<void> makeTeaAsync();
```

```
task<std::string> boilWaterAndMakeTeaAsync()
```

```
{
```

```
    return boilWaterAsync()
```

```
        .then([] {
```

```
            return makeTeaAsync();
```

```
        })
```

```
        .then([] {
```

```
            return "tea ready"s;
```

```
        });
```

```
}
```

Why do I need coroutines?

```
task<void> boilWaterAsync();  
task<void> makeTeaAsync();  
  
task<std::string> boilWaterAndMakeTeaAsync()  
{  
    co_await boilWaterAsync();  
  
    co_await makeTeaAsync();  
  
    co_return "tea ready";  
}
```

```
auto avatar = http::client::http_client{"https://reqres.in"}
    .request(http::methods::GET, "/api/users/1")
    .then([](const http::http_response &response) {
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get user");
        return response.extract_json();
    })
    .then([](const json::value &response) {
        const auto url = response.at("data").at("avatar").as_string();
        return http::client::http_client(url).request(http::methods::GET);
    })
    .then([](const concurrency::task<http::http_response> &result) {
        const auto response = result.get();
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get avatar");
        return response.extract_vector();
    });
```

```
auto avatar = http::client::http_client{"https://reqres.in"}
    .request(http::methods::GET, "/api/users/1")
    .then([](const http::http_response &response) {
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get user");
        return response.extract_json();
    })
    .then([](const json::value &response) {
        const auto url = response.at("data").at("avatar").as_string();
        return http::client::http_client(url).request(http::methods::GET);
    })
    .then([](const concurrency::task<http::http_response> &result) {
        const auto response = result.get();
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get avatar");
        return response.extract_vector();
    });
```

```
const http::http_response userResponse =
    co_await http::client::http_client{"https://reqres.in"}
        .request(http::methods::GET, "/api/users/1");
if (userResponse.status_code() != http::status_codes::OK)
    throw std::runtime_error("Failed to get user");
const json::value jsonResponse = co_await userResponse.extract_json();

const auto url = jsonResponse.at("data").at("avatar").as_string();
const http::http_response avatarResponse =
    co_await http::client::http_client{url}
        .request(http::methods::GET);

if (avatarResponse.status_code() != http::status_codes::OK)
    throw std::runtime_error("Failed to get avatar");
auto avatar = co_await avatarResponse.extract_vector();
```

Why do I need generators?

```
std::generator<std::string> stopGort(  
    std::string suffix) {  
    co_yield "Klaatu" + suffix;  
    co_yield "barada" + suffix;  
    co_yield "nikto" + suffix;  
}
```

```
auto words = stopGort(", please");  
for (auto i : words)  
    std::cout << i << '\n';
```


Why do I need generators?

```
std::generator<int> fibonacci() {  
    for (int cur = 0, next = 1;;) {  
        co_yield cur;  
        cur = std::exchange(next, cur + next);  
    }  
}
```

```
for (auto n : fibonacci())  
    std::cout << n << '\n';
```

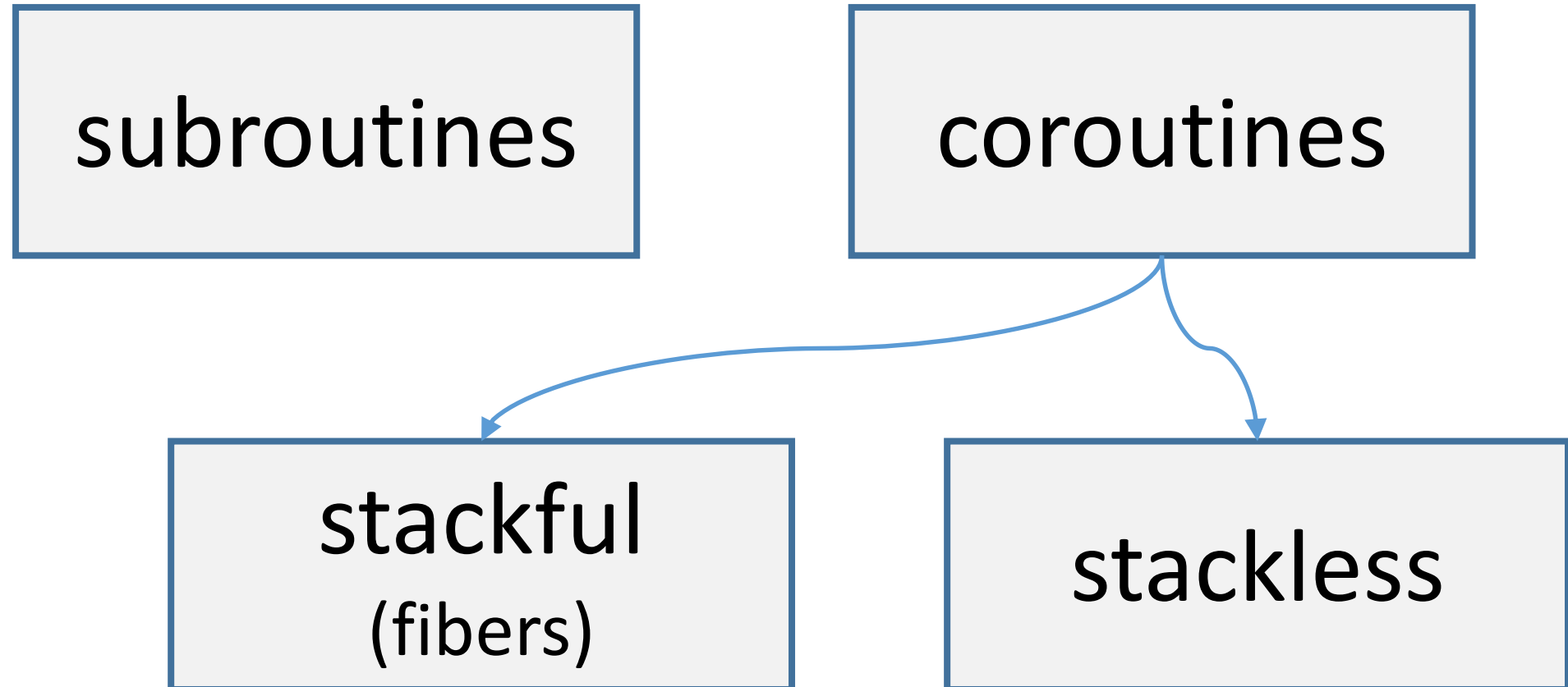
Syntactic sugar?

In large quantities
may cause
syntactic obesity
or even
syntactic diabetes.

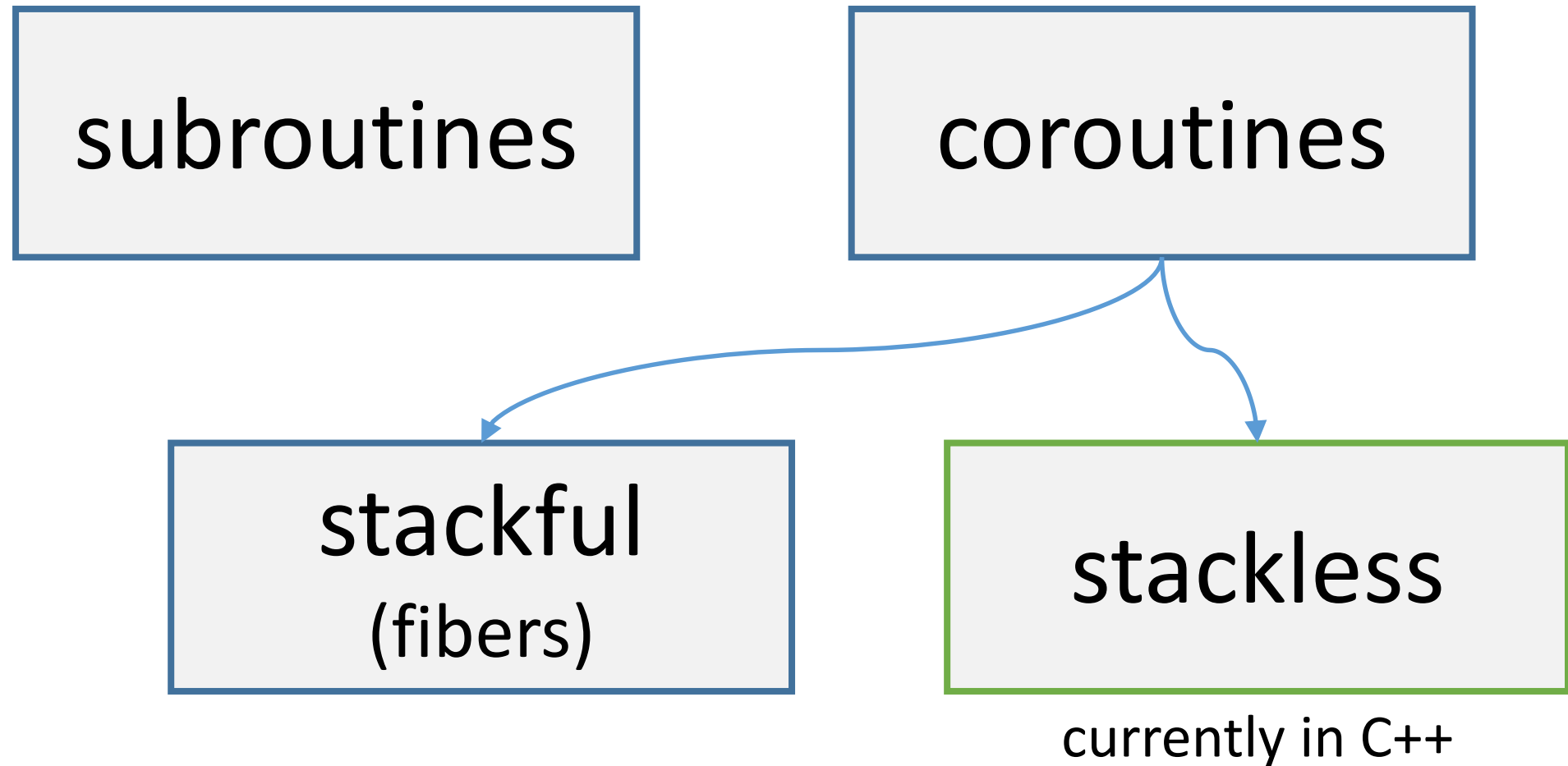
Syntactic sugar?

In large quantities
may cause
syntactic obesity
or even
syntactic diabetes.

What are coroutines?



What are coroutines?



What are coroutines?

Any function in which any of these are used

`co_await`

`co_return`

`co_yield`

Is an implementation detail, does not affect function signature.

What are coroutines?

Any function in which any of these are used

`co_await`

`co_return`

`co_yield`

Is an implementation detail, does not affect function signature.

```
std::future<std::string> makeSandwichesAsync();
```

May be a coroutine, may be not.

What happens "behind the curtains"

```
co_await boilWaterAsync();
```



```
//std::experimental::coroutine_handle<> coroutineHandle;
```

```
auto awaitable = operator co_await(boilWaterAsync());
```

```
if (!awaitable.await_ready()) {
```

```
    awaitable.await_suspend(coroutineHandle);
```

```
    //suspend & resume
```

```
}
```

```
awaitable.await_resume();
```


What happens "behind the curtains"

```
co_await boilWaterAsync();
```

from within the coroutine context



```
//std::experimental::coroutine_handle<> coroutineHandle;
```

```
auto awaitable = operator co_await(boilWaterAsync());
```

```
if (!awaitable.await_ready()) {
```

```
    awaitable.await_suspend(coroutineHandle);
```

```
    //suspend & resume
```

```
}
```

```
awaitable.await_resume();
```

What happens "behind the curtains"

```
co_await boilWaterAsync();
```



```
//std::experimental::coroutine_handle<> coroutineHandle;
```

```
auto awaitable = operator co_await(boilWaterAsync());
```

```
if (!awaitable.await_ready()) {
```

```
    awaitable.await_suspend(coroutineHandle);
```

```
    //suspend & resume
```

```
}
```

```
awaitable.await_resume();
```

schedule continuation

What happens "behind the curtains"

```
co_await boilWaterAsync();
```



```
//std::experimental::coroutine_handle<> coroutineHandle;
```

```
auto awaitable = operator co_await(boilWaterAsync());
```

```
if (!awaitable.await_ready()) {
```

```
    awaitable.await_suspend(coroutineHandle);
```

```
    //suspend & resume
```

```
}
```

```
awaitable.await_resume(),
```

return value

What happens "behind the curtains"


```
task<std::string> boilWaterAndMakeTeaAsync()  
{  
    co_await boilWaterAsync();  
    co_await makeTeaAsync();  
    co_return "tea ready";  
}
```

What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {
    auto p =
        std::coroutine_traits<task<std::string>>::promise_type{};
    auto returnObject = p.get_return_object();
    co_await p.initial_suspend();
    try {
        co_await boilWaterAsync(); //suspend & resume
        co_await makeTeaAsync(); //suspend & resume
        p.return_value("tea ready"); goto final_suspend; //co_return
    }
    catch (...) { p.unhandled_exception(); }
final_suspend:
    co_await p.final_suspend();
}
```

What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {  
    auto p =  
        std::coroutine_traits<task<std::string>>::promise_type{};  
    auto returnObject = p.get_return_object();  
    co_await p.initial_suspend();  
    try {  
        co_await boilWaterAsync(); //suspend & resume  
        co_await makeTeaAsync(); //suspend & resume  
        p.return_value("tea ready"); goto final_suspend; //co_return  
    }  
    catch (...) { p.unhandled_exception(); }  
final_suspend:  
    co_await p.final_suspend();  
}
```



part of coroutine context


What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {  
    auto p =  
        std::coroutine_traits<task<std::string>>::promise_type{};  
    auto returnObject = p.get_return_object();  
    co_await p.initial_suspend();  
    p.resume(); //suspend & resume  
    p.resume(); //suspend & resume  
    p.return_value("tea ready"); goto final_suspend; //co_return  
}  
catch (...) { p.unhandled_exception(); }  
final_suspend:  
    co_await p.final_suspend();  
}
```

returned on
the first suspension

What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {
    auto p =
        std::coroutine_traits<task<std::string>>::promise_type{};
    auto returnObject = p.get_return_object();
    co_await p.initial_suspend();
    try {
        co_await boilWaterAsync(); //suspend & resume
        co_await makeTeaAsync(); //suspend & resume
        p.return_value("tea ready"); goto final_suspend; //co_return
    }
    catch (...) { p.unhandled_exception(); }
final_suspend:
    co_await p.final_suspend();
}
```



What happens "behind the curtains"

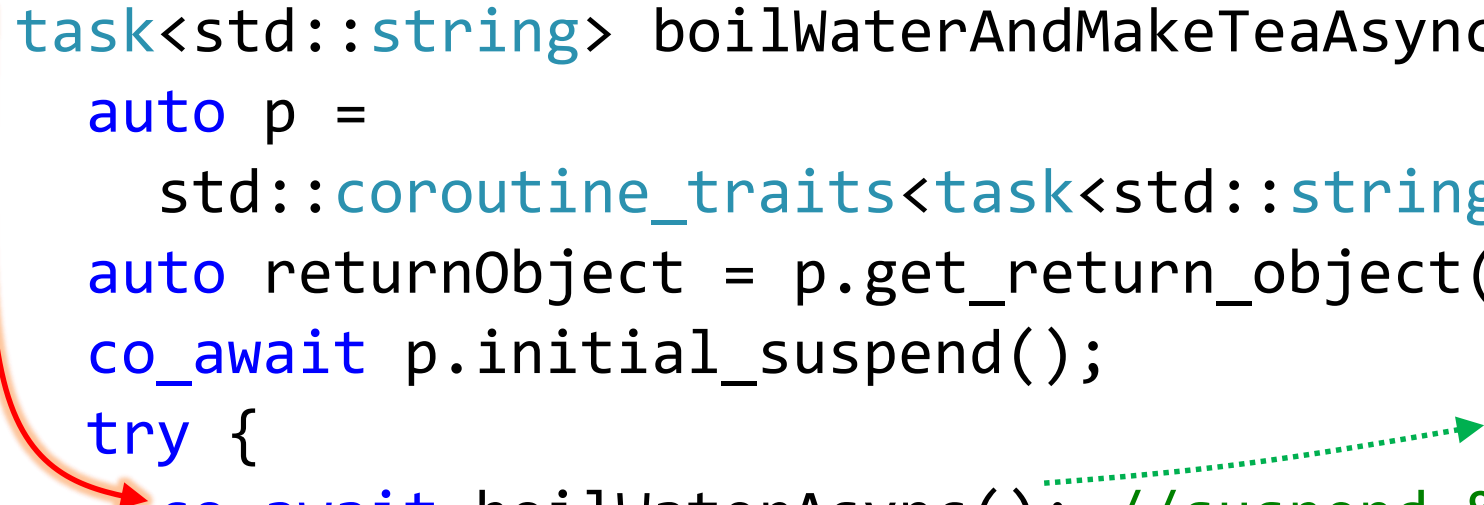
```
task<std::string> boilWaterAndMakeTeaAsync() {  
    auto p =  
        std::coroutine_traits<task<std::string>>::promise_type{};  
    auto returnObject = p.get_return_object();  
    co_await p.initial_suspend();  
    try {  
        ⇒ co_await boilWaterAsync(); //suspend & resume  
        co_await makeTeaAsync(); //suspend & resume  
        p.return_value("tea");  
    }  
    catch (...) { p.unhandled_exception(); }  
    final_suspend:  
    co_await p.final_suspend();  
}
```

Call stack:

```
boilWaterAndMakeTeaAsync$_ResumeCoro$2()  
boilWaterAndMakeTeaAsync$_InitCoro$1()  
boilWaterAndMakeTeaAsync()  
main(int argc, char * * argv)
```

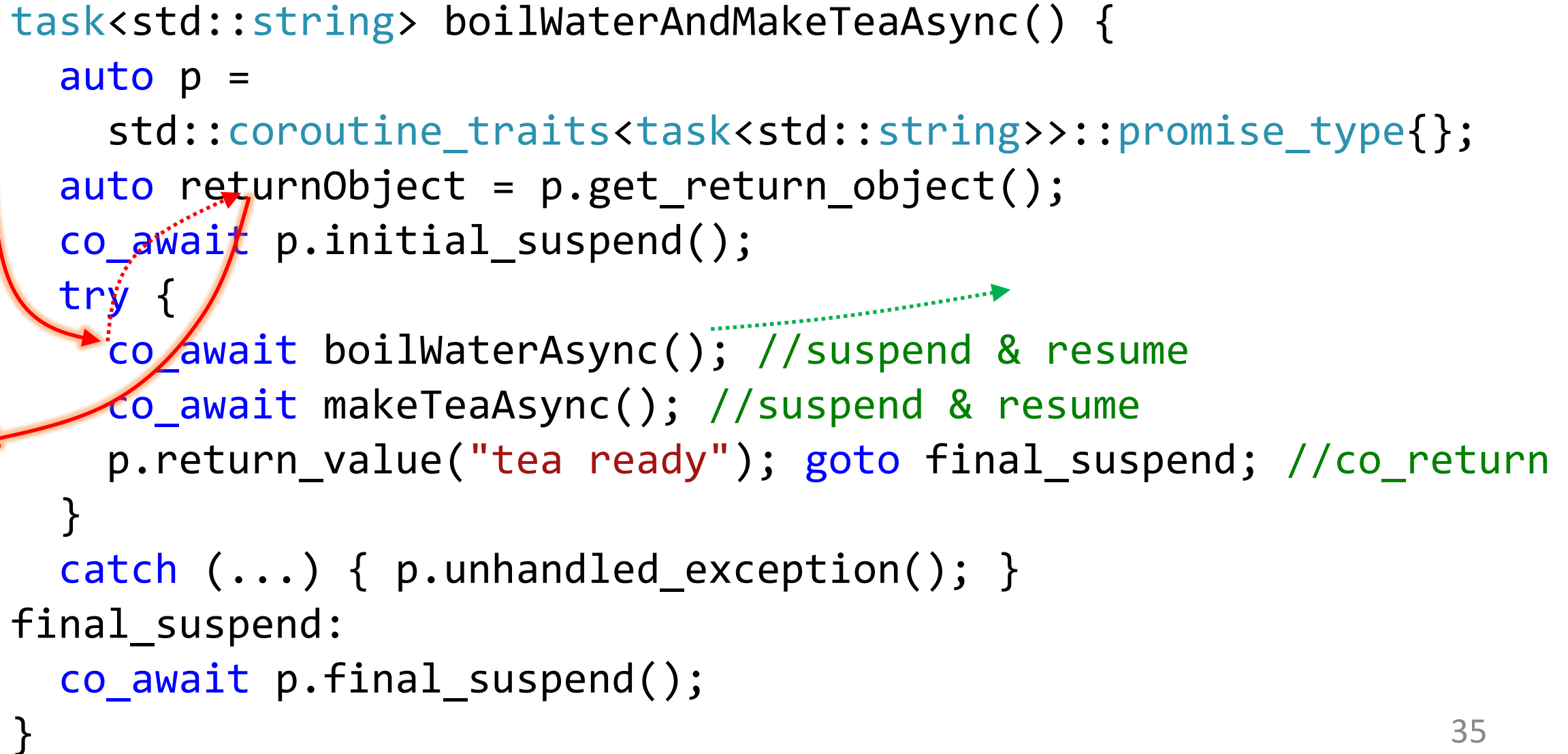
What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {
    auto p =
        std::coroutine_traits<task<std::string>>::promise_type{};
    auto returnObject = p.get_return_object();
    co_await p.initial_suspend();
    try {
        co_await boilWaterAsync(); //suspend & resume
        co_await makeTeaAsync(); //suspend & resume
        p.return_value("tea ready"); goto final_suspend; //co_return
    }
    catch (...) { p.unhandled_exception(); }
final_suspend:
    co_await p.final_suspend();
}
```



What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {  
    auto p =  
        std::coroutine_traits<task<std::string>>::promise_type{};  
    auto returnObject = p.get_return_object();  
    co_await p.initial_suspend();  
    try {  
        co_await boilWaterAsync(); //suspend & resume  
        co_await makeTeaAsync(); //suspend & resume  
        p.return_value("tea ready"); goto final_suspend; //co_return  
    }  
    catch (...) { p.unhandled_exception(); }  
final_suspend:  
    co_await p.final_suspend();  
}
```



What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {
    auto p =
        std::coroutine_traits<task<std::string>>::promise_type{};
    auto returnObject = p.get_return_object();
    co_await p.initial_suspend();
    try {
        co_await boilWaterAsync(); //suspend & resume
        co_await makeTeaAsync(); //suspend & resume
        p.return_value("tea ready"); goto final_suspend; //co_return
    }
    catch (...) { p.unhandled_exception(); }
final_suspend:
    co_await p.final_suspend();
}
```

What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {
    auto p =
        std::coroutine_trait{
            auto returnObject = p;
            co_await p.initial_suspend();
            try {
                co_await boilWaterAndMakeTeaAsync();
                co_await makeTeaAsync(); //suspend & resume
                p.return_value("tea ready"); goto final_suspend; //co_return
            }
            catch (...) { p.unhandled_exception(); }
        }
    final_suspend:
    co_await p.final_suspend();
}
```

Call stack:

```
boilWaterAndMakeTeaAsync$_ResumeCoro$2()
std::coroutine_handle<void>::resume()
std::coroutine_handle<void>::operator>()
<task continuation>
ntdll!TppWorkpExecuteCallback()
ntdll!TppWorkerThread()
```

What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {
    auto p =
        std::coroutine_traits<task<std::string>>::promise_type{};
    auto returnObject = p.get_return_object();
    co_await p.initial_suspend();
    try {
        co_await boilWaterAsync(); //suspend & resume
        co_await makeTeaAsync(); //suspend & resume
        p.return_value("tea ready"); goto final_suspend; //co_return
    }
    catch (...) { p.unhandled_exception(); }
final_suspend:
    co_await p.final_suspend();
}
```

What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {  
    auto p =  
        std::coroutine_traits<task<std::string>>::promise_type{};  
    auto returnObject = p.get_return_object();  
    co_await p.initial_suspend();  
    try {  
        co_await boilWaterAsync(); //suspend & resume  
        co_await makeTeaAsync(); //suspend & resume  
        p.return_value("tea ready"); goto final_suspend; //co_return  
    }  
    catch (...) { p.unhandled_exception(); }  
final_suspend:  
    co_await p.final_suspend();  
}
```

What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {  
    auto p =  
        std::coroutine_traits<task<std::string>>::promise_type{};  
    auto returnObject = p.get_return_object();  
    co_await p.initial_suspend();  
    try {  
        co_await boilWaterAsync(); //suspend & resume  
        co_await makeTeaAsync(); //suspend & resume  
        p.return_value("tea ready"); goto final_suspend; //co_return  
    }  
    catch (...) { p.unhandled_exception(); }  
final_suspend:  
    co_await p.final_suspend();  
}
```

The diagram illustrates the execution flow of the coroutine function. A red arrow points from the function signature to the initial_suspend call. A blue arrow points from the try block back to the final_suspend call. Green arrows show the suspend and resume cycle between the two co_await calls. A blue arrow points from the goto statement to the final_suspend call.

What happens "behind the curtains"

Call stack:

```
boilWaterAndMakeTeaAsync$_ResumeCoro$2()
std::coroutine_handle<void>::resume()
std::coroutine_handle<void>::operator>()
<task continuation>
ntdll!TppWorkpExecuteCallback()
ntdll!TppWorkerThread()
```

```
task<std::string> boilWaterAndMakeTeaAsync() {
    auto p = promise_type::initial_suspend();
    auto returnObject = p.get_return_object();
    co_await p.initial_suspend();
    try {
        co_await boilWaterAndMakeTeaAsync(); //suspend & resume
        p.return_value("tea ready"); goto final_suspend; //co_return
    } catch (...) { p.unhandled_exception(); }
final_suspend:
    co_await p.final_suspend();
}
```

What happens "behind the curtains"

```
task<std::string> boilWaterAndMakeTeaAsync() {  
    auto p =  
        std::coroutine_traits<task<std::string>>::promise_type{};  
    auto returnObject = p.get_return_object();  
    co_await p.initial_suspend();  
    try {  
        co_await boilWaterAsync(); //suspend & resume  
        co_await makeTeaAsync(); //suspend & resume  
        p.return_value("tea ready"); goto final_suspend; //co_return  
    }  
    catch (...) { p.unhandled_exception(); }  
final_suspend:  
    co_await p.final_suspend();  
}
```

What happens "behind the curtains"

```
co_yield expression;
```



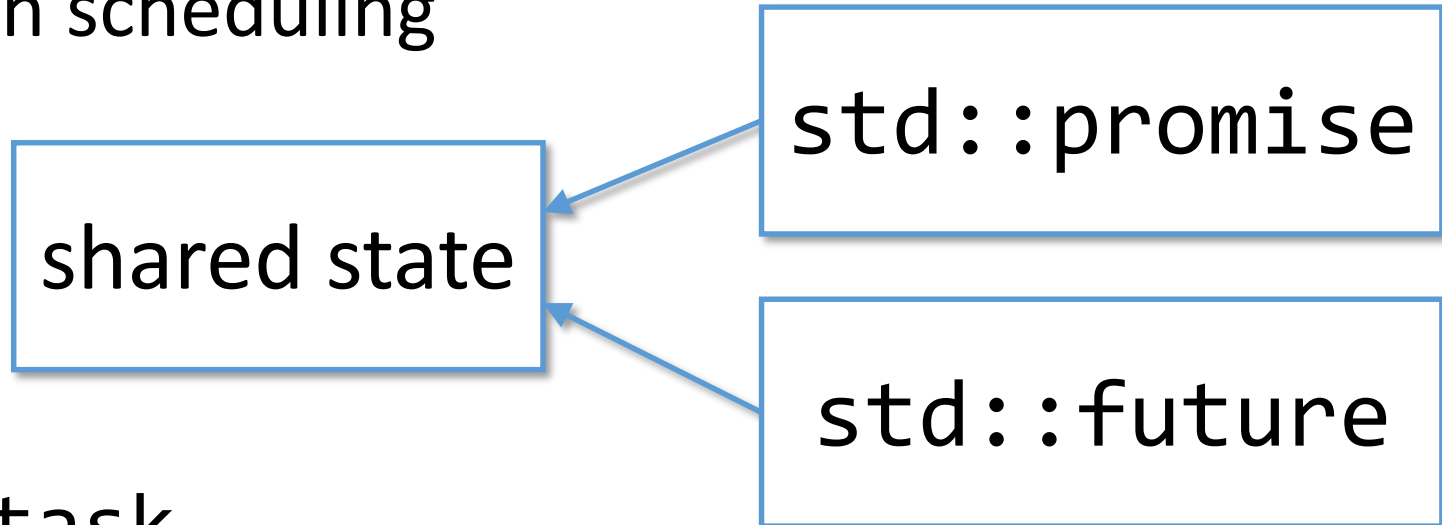
```
co_await promise.yield_value(expression);
```

Gotta go faster



`std::future` is inefficient

- memory allocation for shared state
- synchronization (ref count, get/set)
- overhead of continuation scheduling

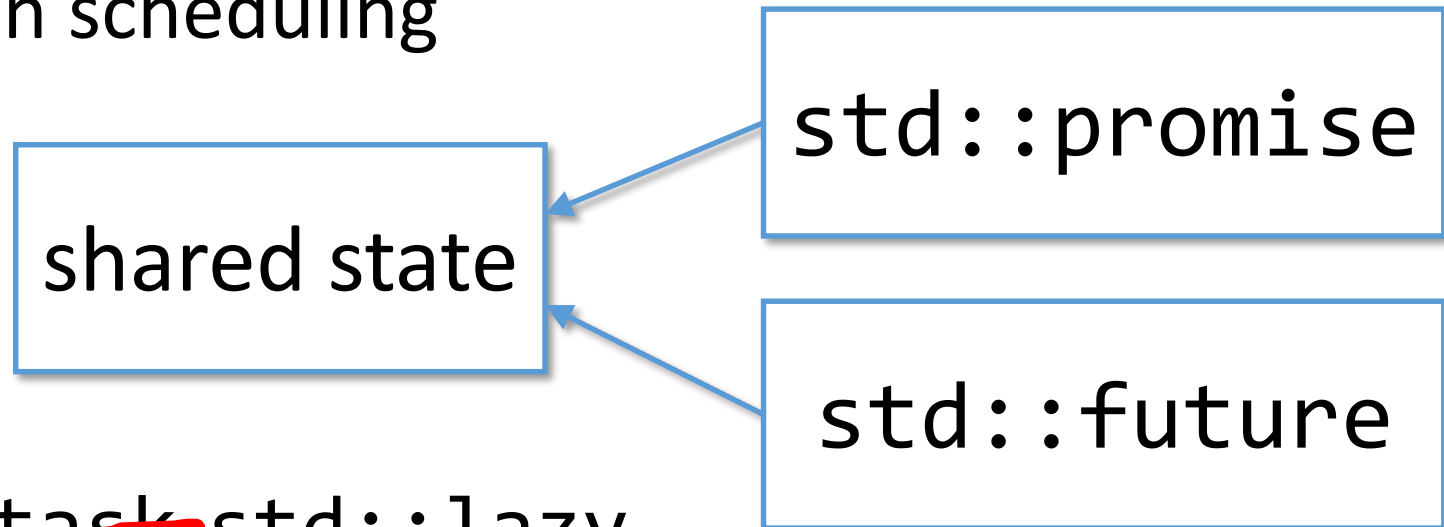


proposal [P1056](#) in
the C++ standard: `std::task`

<https://github.com/lewissbaker/cppcoro>

`std::future` is inefficient

- memory allocation for shared state
- synchronization (ref count, get/set)
- overhead of continuation scheduling



proposal [P1056](#) in
the C++ standard: ~~`std::task`~~ `std::lazy`

<https://github.com/lewissbaker/cppcoro>

Benchmark: `std::future`

```
void future(benchmark::State& state) {
    for (auto i : state) {
        std::future<void> water = std::async(std::launch::deferred, [] {
            boilWater();
        });
        auto tea = std::async(std::launch::deferred,
            [water = std::move(water)]() mutable {
                water.get();
                makeTea();
            });
        tea.get();
    }
}
BENCHMARK(future);
```

time=541 ns
speed=1x

Benchmark: concurrency::task

```
void concurrencyTask(benchmark::State& state) {  
    for (auto i : state) {  
        [] {  
            boilWater();  
            return concurrency::task_from_result();  
        }()  
        .then([] {  
            makeTea();  
        })  
        .wait();  
    }  
}  
BENCHMARK(concurrencyTask);
```

time=7195 ns
speed=0,08x

Benchmark: lightweight Task

```
Task boilWaterAsync() { boilWater(); co_return; }  
Task makeTeaAsync() { makeTea(); co_return; }
```

```
void coroutines(benchmark::State& state) {  
    [&state]() -> std::future<void> {  
        for (auto i : state) {  
            co_await boilWaterAsync();  
            co_await makeTeaAsync();  
        }  
    }().wait();  
}
```

```
BENCHMARK(coroutines);
```

time=204 ns
speed=2,7x

Benchmark

Run on (4 X 3392 MHz CPU s)

CPU Caches:

L1 Data 32K (x4)

L1 Instruction 32K (x4)

L2 Unified 262K (x4)

L3 Unified 6291K (x1)

Benchmark	Time	CPU	Iterations
future	541 ns	547 ns	1000000
concurrencyTask	7195 ns	7254 ns	112000
coroutines	204 ns	204 ns	3446154
rawCalls	1 ns	1 ns	1000000000

lightweight Task

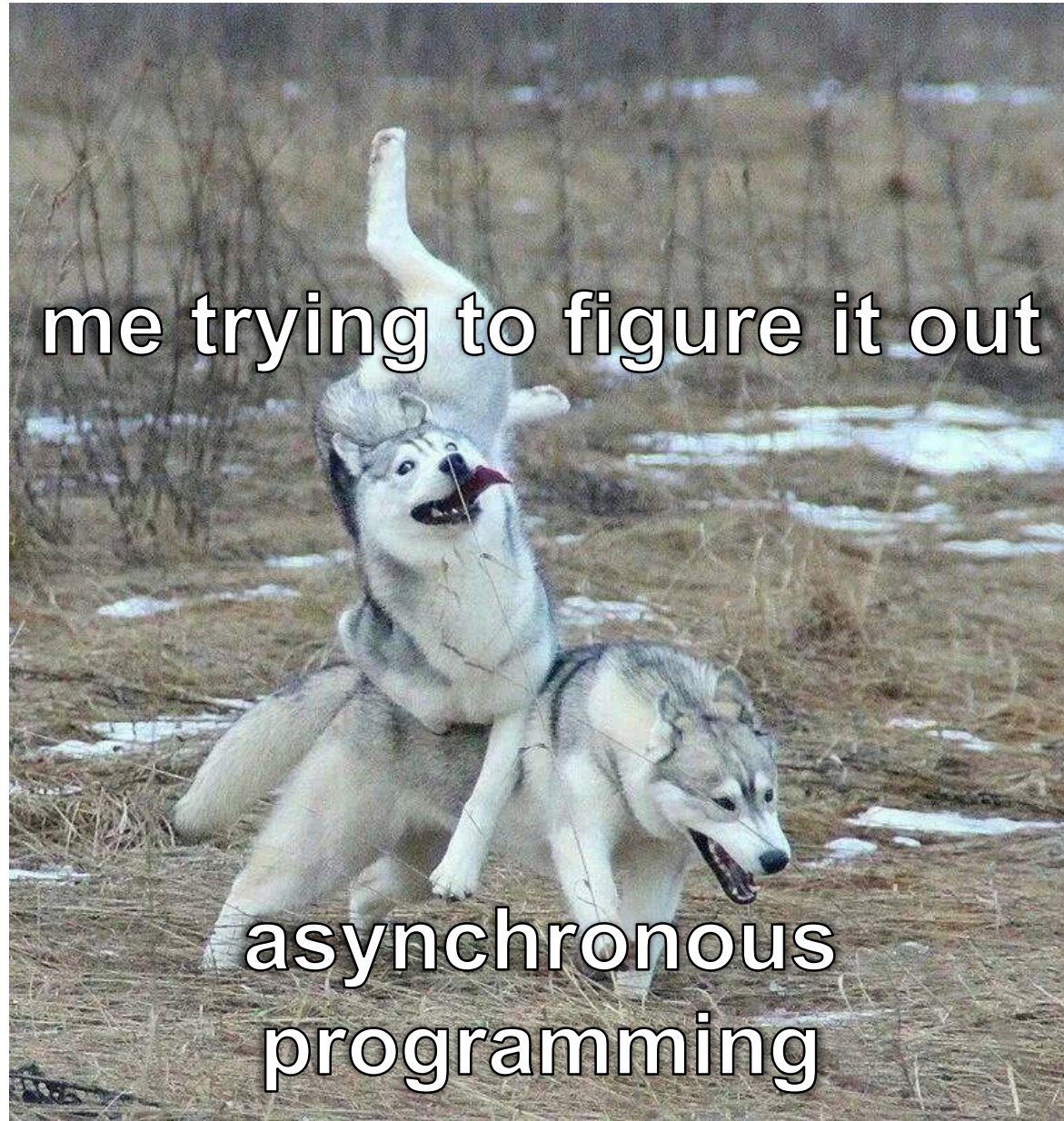
```
struct TaskPromise {
    struct Task get_return_object();
    bool initial_suspend() { return false; }
    auto final_suspend() {
        struct Awaitable {
            bool await_ready() { return false; }
            void await_suspend(std::coroutine_handle<TaskPromise> coro) {
                if (auto continuation = coro.promise().continuation)
                    continuation.resume();
            }
            void await_resume() {}
        };
        return Awaitable{};
    }
    void unhandled_exception() { exception = std::current_exception(); }
    void return_void() {}
    void result() { if (exception) std::rethrow_exception(exception); }
    std::coroutine_handle<> continuation;
    std::exception_ptr exception;
};
```

lightweight Task

```
struct [[nodiscard]] Task {
    using promise_type = TaskPromise;
    Task(coroutine_handle<TaskPromise> coro) : m_coro(coro) {}
    ~Task() { if (m_coro) m_coro.destroy(); }
    friend auto operator co_await(const Task &t) {
        struct Awaitable {
            bool await_ready() { return coro.done(); }
            void await_suspend(coroutine_handle<> coro)
            { this->coro.promise().continuation = coro; }
            void await_resume() { coro.promise().result(); }
            coroutine_handle<TaskPromise> coro;
        };
        return Awaitable{ t.m_coro };
    }
private:
    coroutine_handle<TaskPromise> m_coro;
};

Task TaskPromise::get_return_object() {
    return Task{ coroutine_handle<TaskPromise>::from_promise(*this) };
}
```

Thanks for coming!



Asynchronous C++ programming

Pavel Novikov

 @cpp_ape

Align Technology R&D

align

Slides: <https://git.io/Jew2Y>

