

# Асинхронная разработка на C++

Павел Новиков

[pnovikov@aligntech.com](mailto:pnovikov@aligntech.com)

R&D Align Technology

The logo for Align Technology, featuring the word "align" in a lowercase, sans-serif font. The letter "i" is stylized with a blue dot above it.

# О чём речь

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

# О чём речь

- `std::async`, `std::future`,  
`std::promise`,  
`std::packaged_task`
- PPL: `concurrency::task`,  
`continuations`, `cancellation`,  
`task composition`
- корутины:  
`co_await`, `co_return`,  
генераторы,  
детали реализации

# О чём речь

- `std::async`, `std::future`,  
`std::promise`,  
`std::packaged_task`
- PPL: `concurrency::task`,  
`continuations`, `cancellation`,  
`task composition`
- корутины:  
`co_await`, `co_return`,  
генераторы,  
детали реализации
- будущее?

# Метафора

Неудачная метафора подобна котёнку с дверцей

Интернет

# Метафора

Синхронный вариант:

```
boilWater();
```

```
makeTea();
```

```
drinkTea();
```

# Метафора

Асинхронный вариант:

```
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



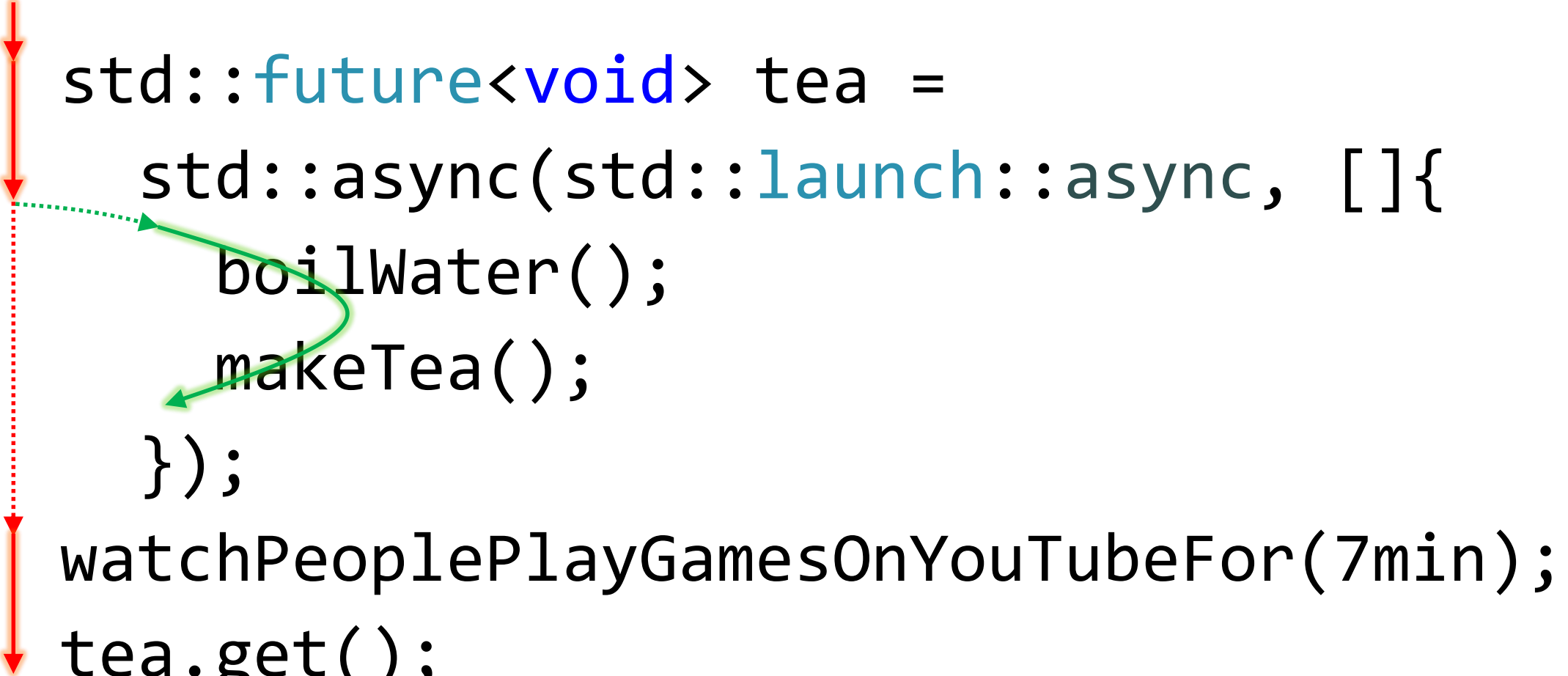
```
std::future<void> tea =  
    std::async(std::launch::async, []{  
        boilWater();  
        makeTea();  
    });  
watchPeoplePlayGamesOnYouTubeFor(7min);  
tea.get();  
drinkTea();
```

# 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();
```

The diagram illustrates the execution flow of the provided C++ code. A vertical red arrow on the left indicates the main thread's execution path. A green arrow shows the execution of the asynchronous lambda function, which runs in parallel to the main thread. The lambda function contains two sequential calls: `boilWater();` and `makeTea();`. The main thread continues to execute `watchPeoplePlayGamesOnYouTubeFor(7min);` and then `tea.get();`, which waits for the lambda function to complete. Finally, the main thread executes `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();
```

The diagram illustrates the execution flow of the code. A red dashed arrow on the left side points downwards, indicating the sequence of execution. A green dotted arrow starts at the task definition, loops back to the execute call, and then points to the future.get() call, showing the flow of control and data between these components.

# 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.

# Идиома: асинхронное обновление значения

```
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

- выпущена Microsoft вместе с Visual Studio 2010 (+ лямбды)
- подмножество (PPLX) реализовано в кроссплатформенной библиотеке 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

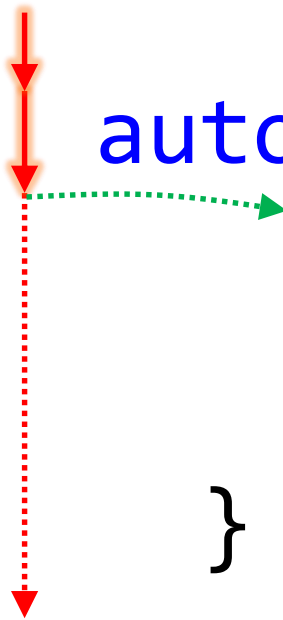
```
task<void> makeTeaAsync();
```

```
↓  
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();
```

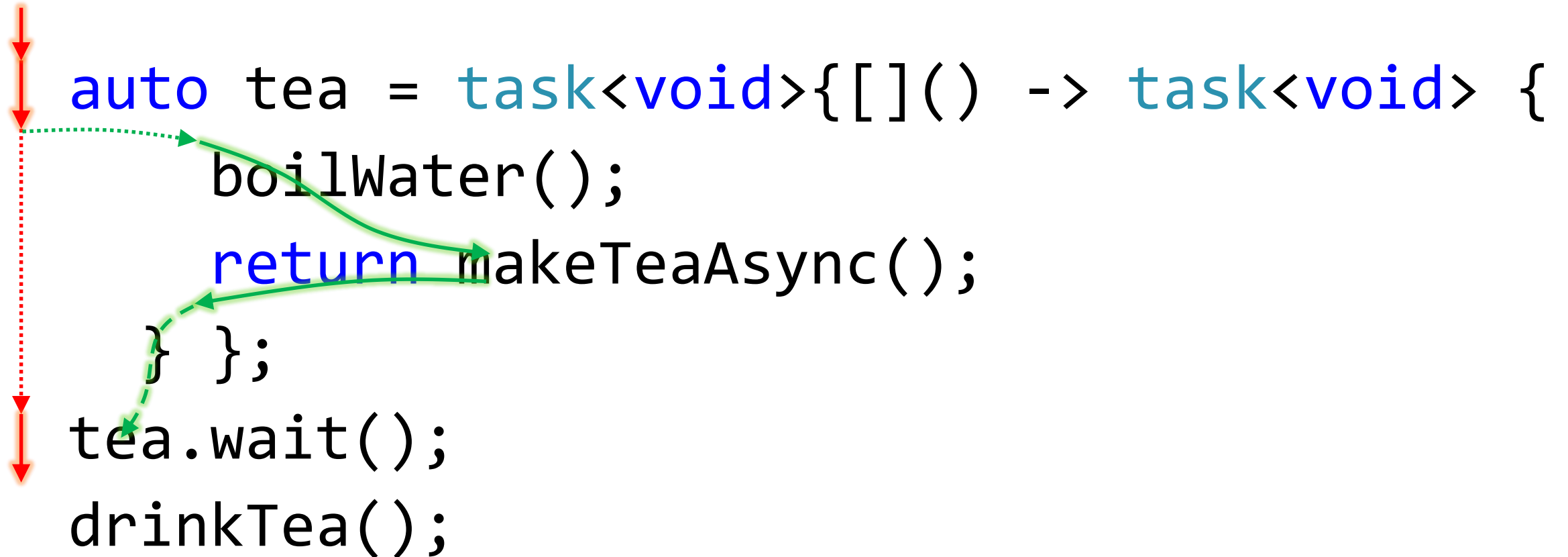
```
auto tea = task<void>{[]() -> task<void> {  
    boilWater();  
    return makeTeaAsync();  
} };  
tea.wait();  
drinkTea();
```

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 back to the lambda function's body. A red dashed arrow also points from the lambda function's return type to the `tea.wait()` call.

# 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 expression to the 'return' statement, and another red dashed arrow points from the lambda expression to the 'wait()' call. A green arrow points from the 'return' statement to the 'makeTeaAsync()' call, and a blue dotted arrow points from the 'makeTeaAsync()' call to the 'wait()' call.



# Task unwrapping

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

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

# Continuations

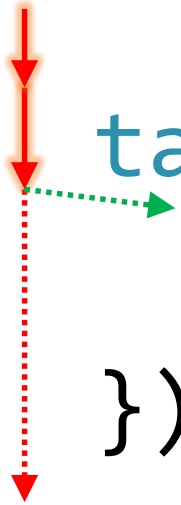


```
task<std::string> water = create_task([] {  
    boilWater(); return "Water boiled"s;  
});  
task<void> tea = water.then(  
    [](const std::string &msg) {  
        makeTea();  
    });  
tea.wait();
```

# 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

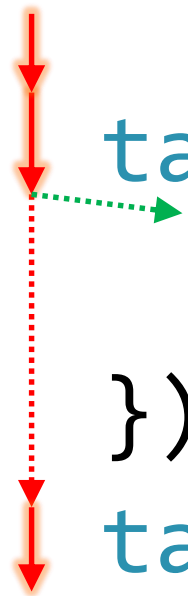


```
task<std::string> water = create_task([] {
    boilWater(); return "Water boiled"s;
});
task<void> tea = water.then(
    [](const std::string &msg) {
        makeTea();
    });
tea.wait();
```

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();
```

# 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 execution flow of the code. A vertical red dashed line with downward-pointing arrows indicates the main thread's execution path. It starts at the top, passes through the opening curly brace of the `water` task, then through the `boilWater()` call, and then through the closing curly brace of the `water` task. It then passes through the opening curly brace of the `tea` task, then through the `makeTea()` call, and finally through the closing curly brace of the `tea` task. A green solid arrow starts at the opening curly brace of the `water` task and points to the `boilWater()` call. Another green solid arrow starts at the `return` statement and points back to the opening curly brace of the `water` task, representing the continuation of the `water` task. A green dotted arrow starts at the opening curly brace of the `water` task and points to the opening curly brace of the `tea` task, representing the continuation of the `main` thread.



# Continuations

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

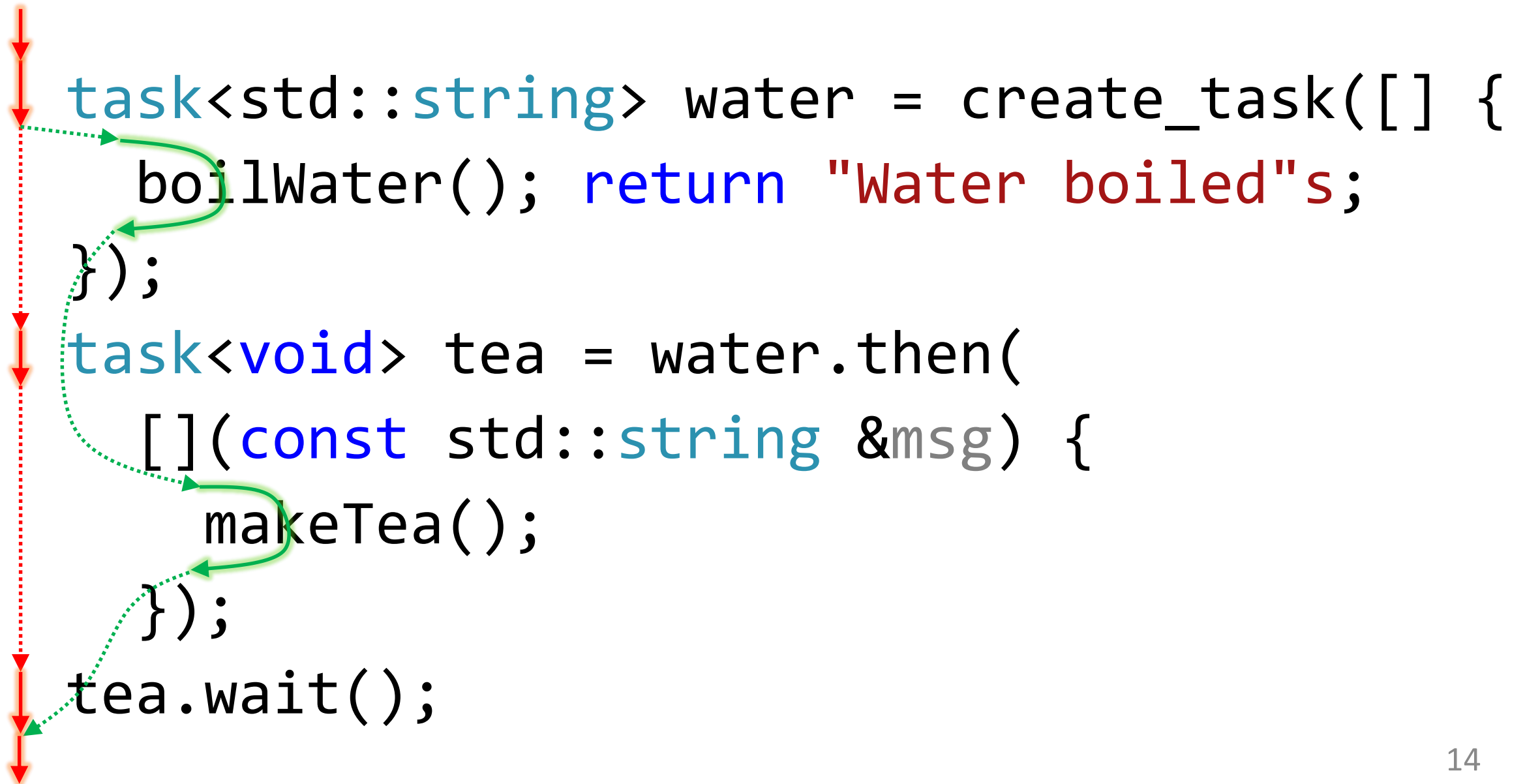
The diagram illustrates the execution flow of the code. A vertical red dashed line with downward-pointing arrows indicates the main thread's execution path. It starts at the top, moves down to the opening curly brace of the `water` task, then continues down past the closing curly brace of `water` to the opening curly brace of the `tea` task, and finally reaches the `tea.wait()` statement. Green curved arrows represent continuations: one starts at the `return` statement in the `water` task and points to the opening curly brace of the `tea` task, and another starts at the `makeTea()` statement and points to the closing 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 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 continues down past the `boilWater()` call and the `return` statement, reaching the closing curly brace. From there, it moves down to the opening curly brace of the `tea` task, then continues down past the `makeTea()` call and the closing curly brace. Finally, it reaches the `tea.wait()` call. Green curved arrows show the continuation of control: one arrow starts at the `return` statement in the `water` task and points to the opening curly brace of the `tea` task, and another arrow starts at the `makeTea()` call and points to the closing curly brace of the `tea` task.

# Continuations



# Обработка исключений



```
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();
```


# Обработка исключений



```
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();
```

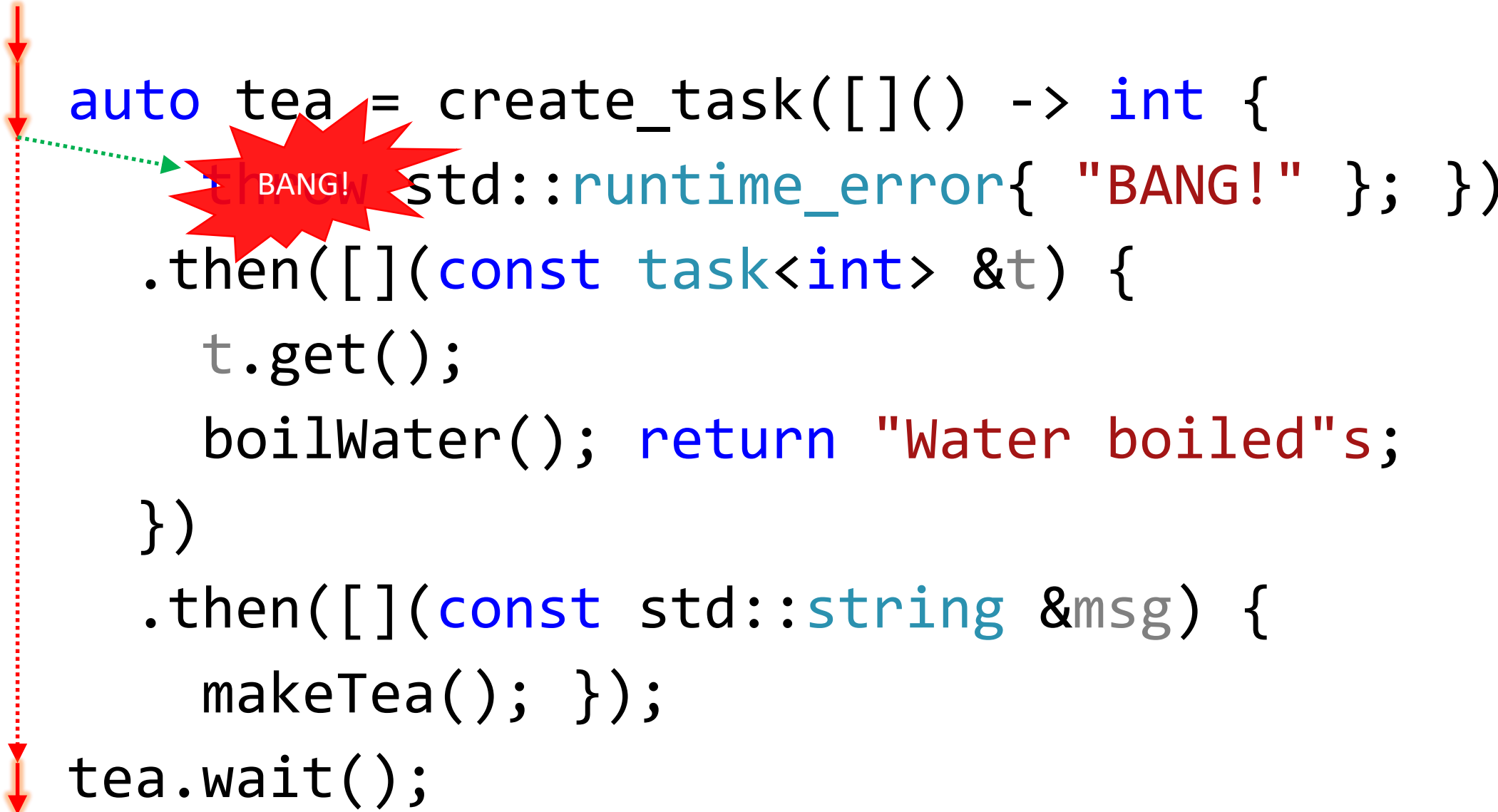
# Обработка исключений

```
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();
```

A vertical red dashed line with downward-pointing arrowheads at the top and bottom, indicating the flow of an exception from the `throw` statement in the first lambda function down to the `wait()` call at the end of the program. A green dotted arrow points from the `throw` statement to the right.

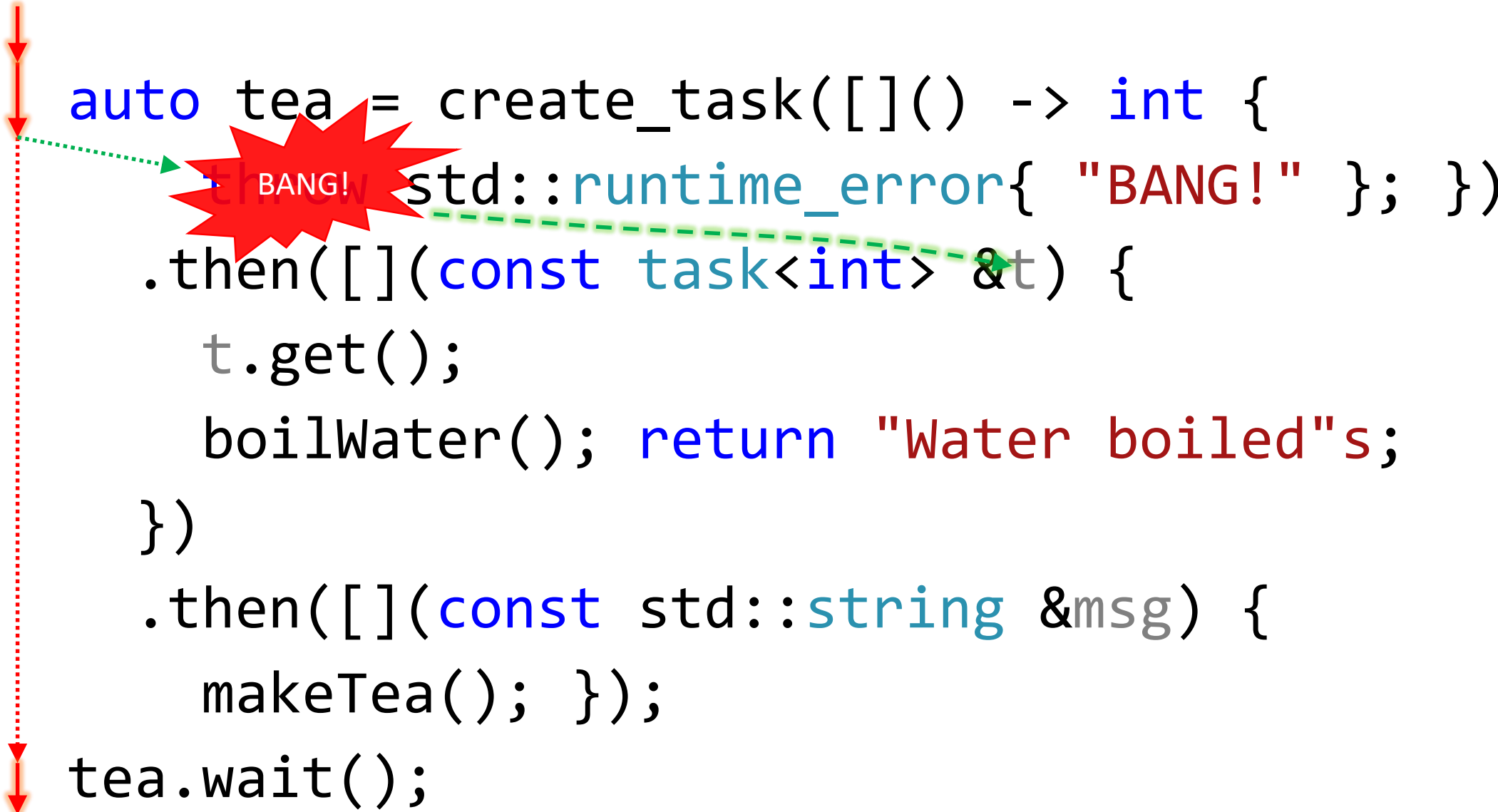
# Обработка исключений

```
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();
```



# Обработка исключений

```
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();
```



The diagram features several annotations on the code:

- A red starburst containing the word "BANG!" is positioned over the `throw` statement in the first lambda function.
- A red dashed arrow points from the top left towards the `throw` statement.
- A green dashed arrow points from the `throw` statement to the `t.get()` call in the second lambda function.
- A red dashed arrow points from the top left towards the `tea.wait()` call at the bottom of the code.

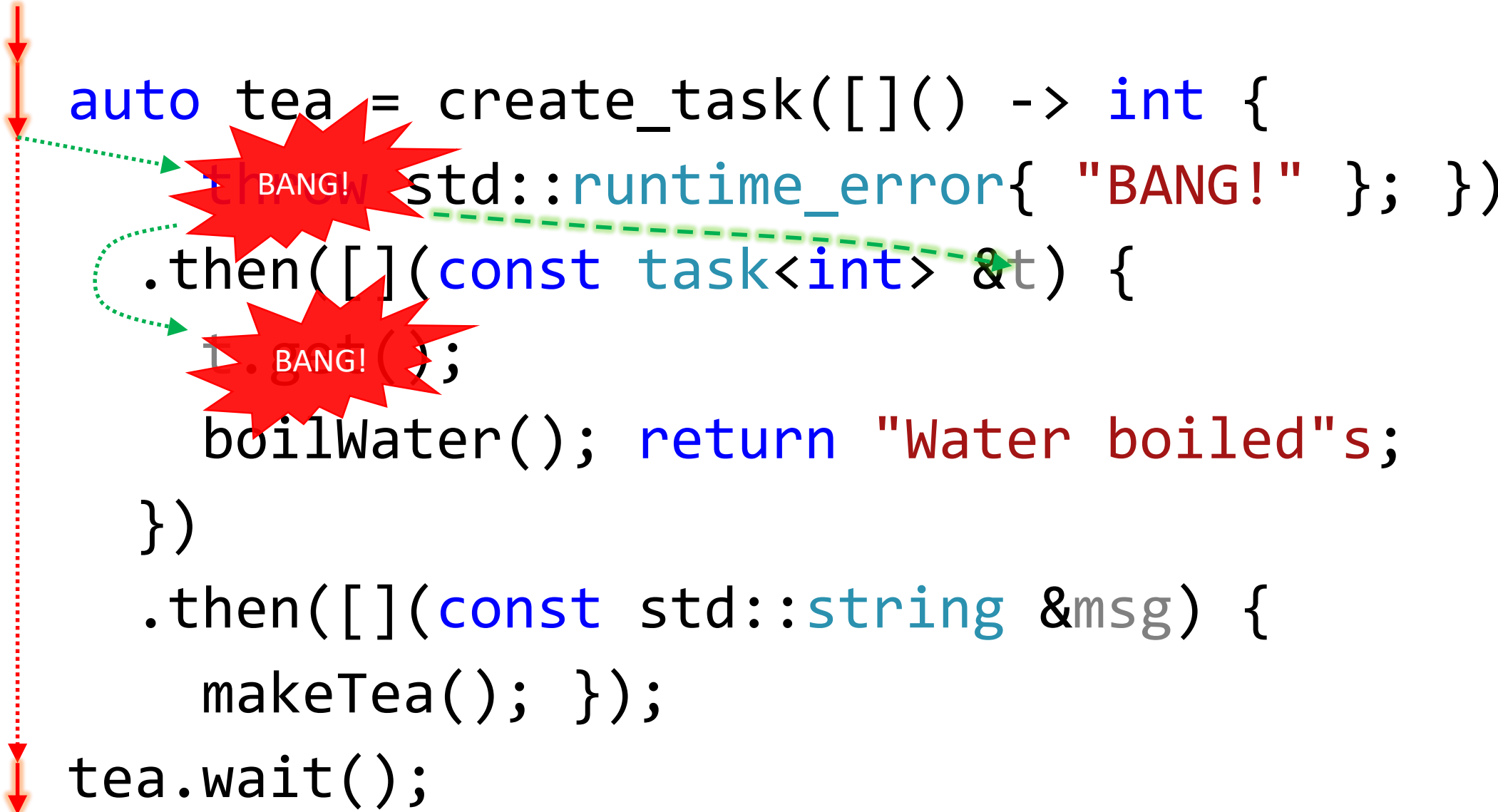


# Обработка исключений

```
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();
```

# Обработка исключений

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



# Обработка исключений

```
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();
```

# Обработка исключений

```
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();
```

# Обработка исключений

```
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();
```

Использование:

```
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);
}
```

предложение [P0660](#)  
в стандарт C++

## 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()  
};
```

возвращает результаты задач



```
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()  
};
```

возвращает результат задачи и её индекс



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

# Идиома: 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();
        });
}
```

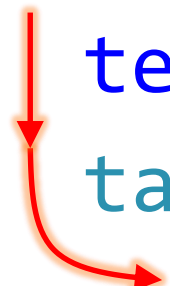


## Идиома: do while

```
↓
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();
        });
}
```

## Идиома: do while

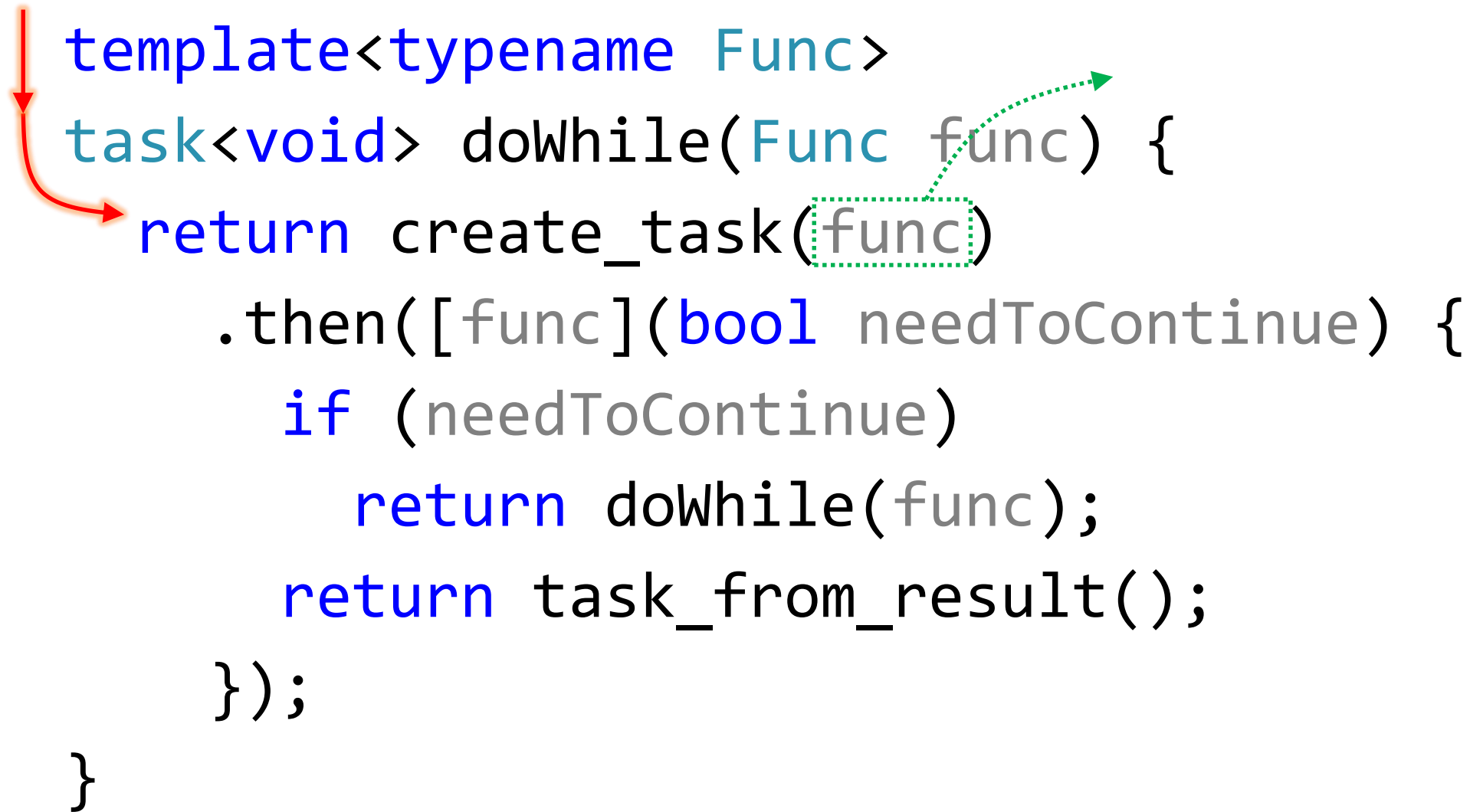
```
template<typename Func>
task<void> doWhile(Func func) {
    return create_task(func)
        .then([func](bool needToContinue) {
            if (needToContinue)
                return doWhile(func);
            return task_from_result();
        });
}
```





## Идиома: 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 flow of control in the provided C++ code. A red arrow starts at the opening curly brace of the `doWhile` function and points to the `return` statement. A green dashed arrow starts from the `func` parameter in the `doWhile` function signature and points to the `func` argument in the `create_task` call. A green dashed box highlights the `func` argument in the `create_task` call.

## Идиома: 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` call, and finally to the `.then` lambda function. A green dashed box highlights the `func` parameter passed to `create_task`, with a green dashed arrow pointing from it to the `[func]` capture list in the `.then` lambda, showing how the function object is captured for later use.

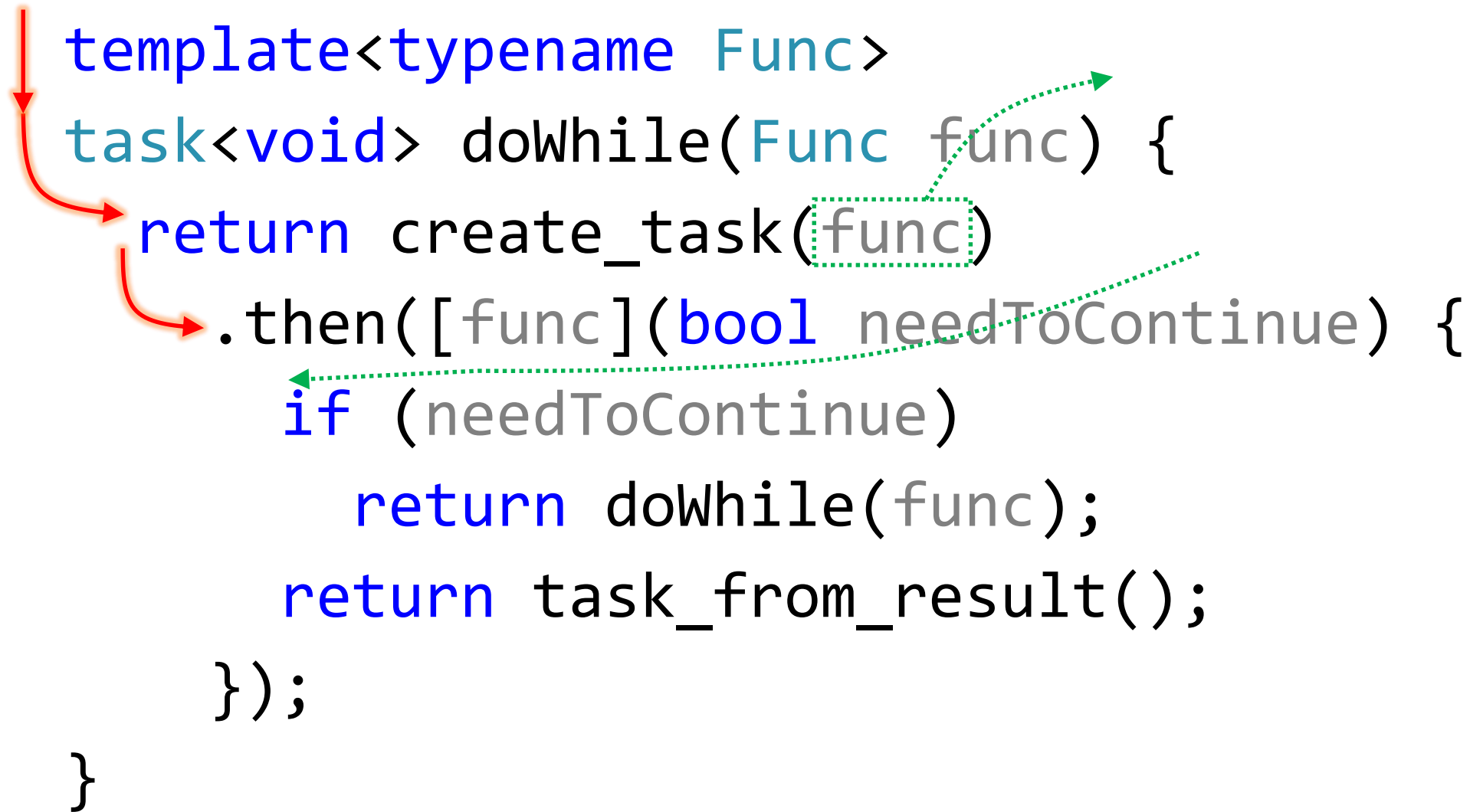
## Идиома: do while

```
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: from the start of the function, down to the `return` statement, then to the `create_task` function, and finally to the `.then` lambda function. Green arrows and boxes highlight a recursive loop: a green arrow points from the `func` argument in `create_task(func)` to the `func` parameter in the `.then` lambda; another green arrow points from the `needToContinue` parameter to the `if` statement; a third green arrow points from the `return doWhile(func);` line back to the `if` statement, indicating a recursive call. A green dashed box encloses the `if` statement and the `return` lines, representing the loop body.

## Идиома: 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 execution path: from the start of the function, down to the `return` statement, then to the `create_task` function call, and finally to the `.then` lambda function. Green dotted arrows show the recursive loop: one arrow points from the `func` argument in `create_task` to the `[func]` capture in the `.then` lambda; another arrow points from the `if` condition to the recursive `doWhile` call; and a third arrow points from the recursive call back to the `if` condition, forming a loop.

## Идиома: 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 recursive nature of the `doWhile` function. Red arrows indicate the call flow: from the `doWhile` function call to `create_task(func)`, then to the `then` block, and finally to the recursive `doWhile` call. Green dotted arrows show the return path: from the recursive `doWhile` call back to the `then` block, and then to the `create_task` function.

# Идиома: 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 recursive nature of the `doWhile` function. Red arrows show the flow of control: from the `doWhile` function call to the `return` statement, then to the `then` block, and finally to the `if` block. A green arrow highlights the recursive call: it starts from the `if` block, loops back to the `return` statement, and then to the `doWhile` function call, indicating that the function calls itself when `needToContinue` is true.

# Идиома: 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 'do while' idiom in C++ using a lambda function. Red arrows show the flow of control: from the function signature to the lambda call, then to the lambda body, and finally to the return statement. A green arrow shows a recursive call from the lambda body back to the doWhile function. A dotted green box highlights the 'func' parameter in the lambda call, and a dotted green arrow points from it to the lambda body, indicating that the lambda captures the function by reference.

# Идиома: 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();
    });
}
```

Diagram illustrating the recursive implementation of a do-while loop using C++ tasks. Red arrows show the flow of control from the function call to the recursive call. Green arrows show the return path from the recursive call back to the function call.





# Идиома: отмена нескольких запросов

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

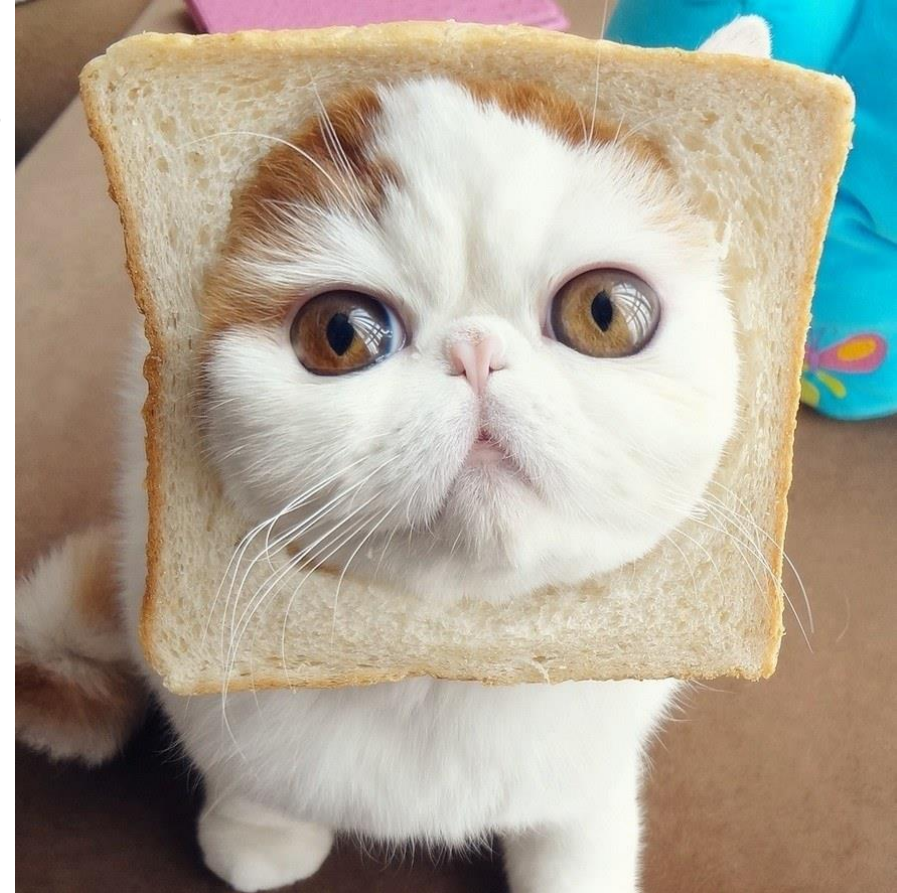
struct Gadget {
    task<std::string> makeRequest(const std::string &request) {
        return makeRequest(request, m_tokenSource.get_token());
    }
    void cancelAllRequests() {
        m_tokenSource.cancel();
        m_tokenSource = {};
    }
private:
    cancellation_token_source m_tokenSource;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

# Идиома: continuation chaining

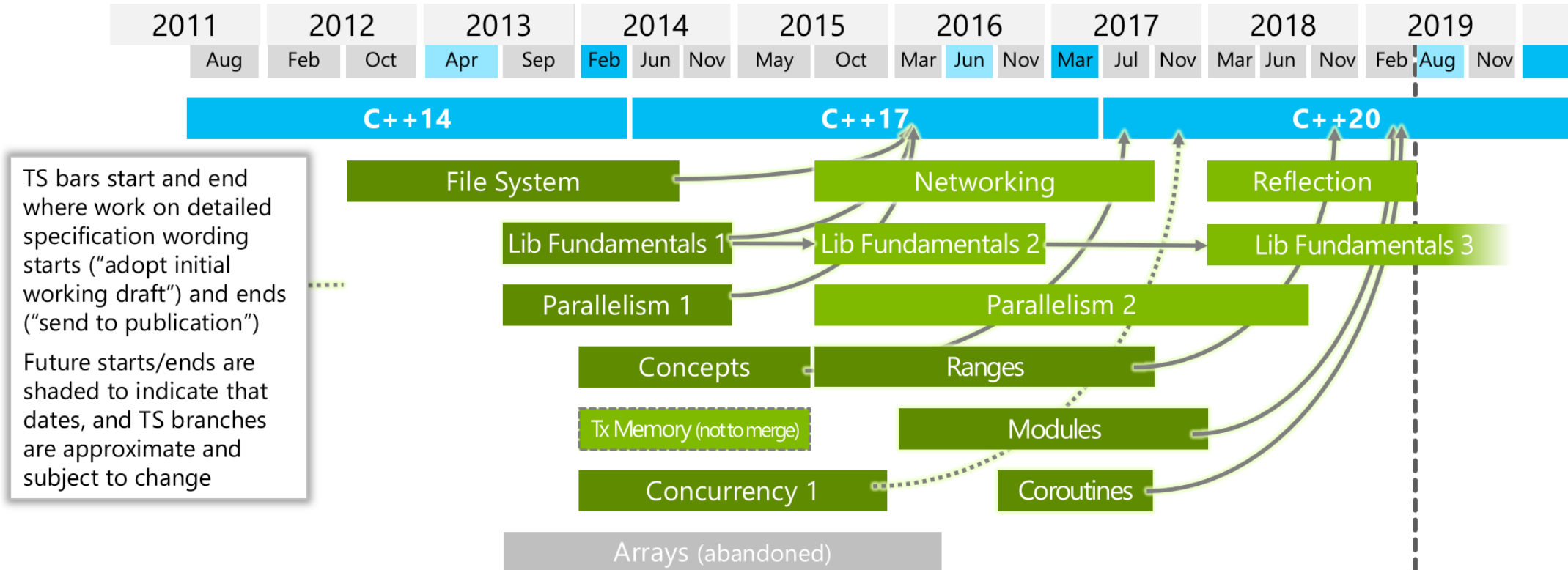
```
auto avatar = http::client::http_client{"https://reqres.in"}
    .request(http::methods::GET, "/api/users/1")
    .then([](const http::http_response &response) {
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get user");
        return 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();
    });
```

# Идиома: continuation chaining

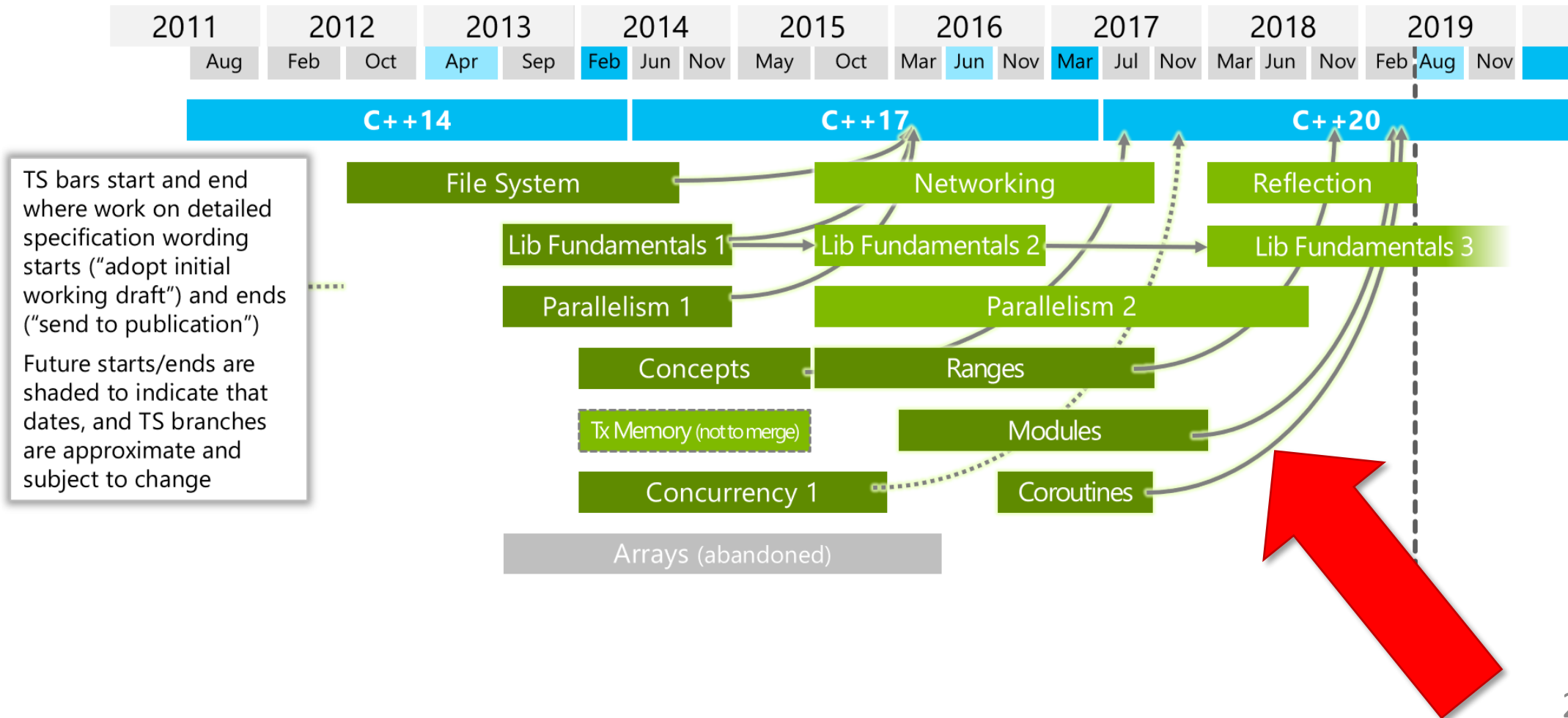
```
auto avatar = http::client::http_client{"https://reqres.in"}
    .request(http::methods::GET, "/api/users/1")
    .then([](const http::http_response &response) {
        if (response.status_code() != http::status_codes::OK)
            throw std::runtime_error("Failed to get user");
        return 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();
    });
```



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



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



# Зачем мне корутины?

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

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

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

```
{
```

```
    return boilWaterAsync()
```

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

```
            return makeTeaAsync();
```

```
        })
```

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

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

```
        });
```

```
}
```

# Зачем мне корутины?

```
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();
    });

```



```
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();
```

# Зачем мне генераторы?

```
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';
```

# Зачем мне генераторы?

```
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';
```

# Синтаксический сахар?

В больших количествах  
может вызвать  
синтаксическое ожирение  
или даже  
синтаксический диабет.

# Синтаксический сахар?

В больших количествах  
может вызвать  
синтаксическое ожирение  
или даже  
синтаксический диабет.

# Что такое корутины?

Любая функция, в которой используется хотя бы одно из

`co_await`

`co_return`

`co_yield`

Является деталями реализации, не влияет на сигнатуру функции.

# Что такое корутины?

Любая функция, в которой используется хотя бы одно из

`co_await`

`co_return`

`co_yield`

Является деталями реализации, не влияет на сигнатуру функции.

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

Может быть корутиной, может не быть.

Что происходит "за кулисами"

```
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();
```



# Что происходит "за кулисами"

```
co_await boilWaterAsync();
```



из контекста корутины



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

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

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

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

```
    //suspend & resume
```

```
}
```

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

# Что происходит "за кулисами"

```
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();
```

запланировать продолжение

# Что происходит "за кулисами"

```
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(),
```

вернуть значение

Что происходит "за кулисами"

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

# Что происходит "за кулисами"

```
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();
}
```

# Что происходит "за кулисами"

```
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();  
}
```

часть контекста корутины


# Что происходит "за кулисами"

```
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();  
}
```

возвращается на  
первой приостановке

# Что происходит "за кулисами"

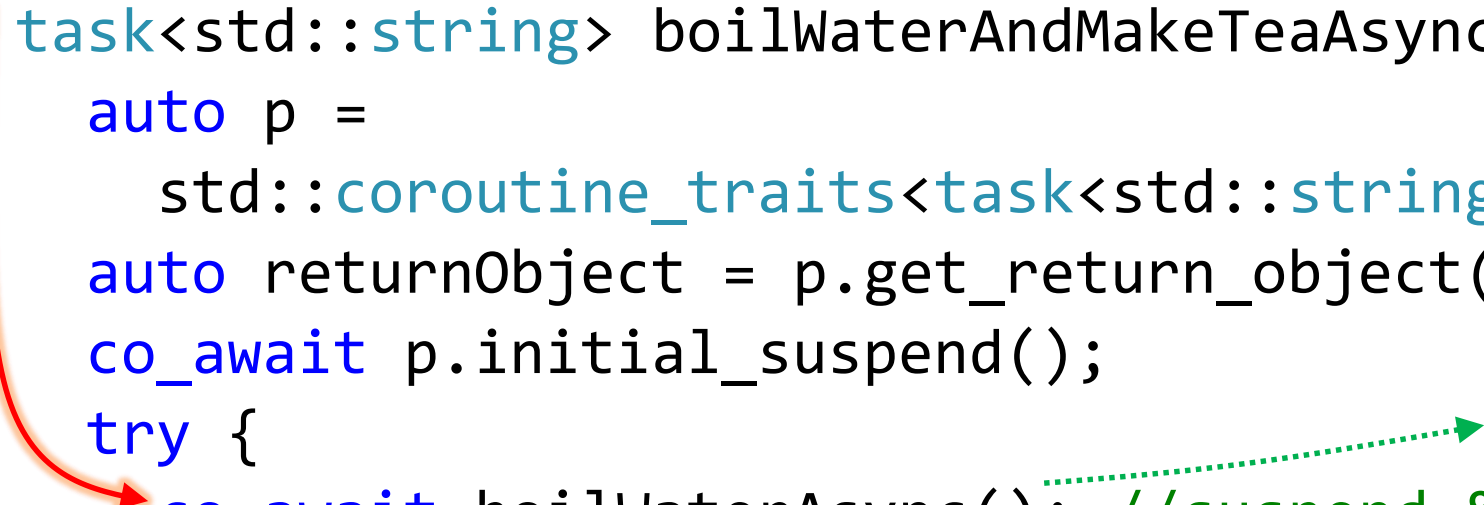
```
task<std::string> boilWaterAndMakeTeaAsync() {
    auto p =
        std::coroutine_traits<task<std::string>>::promise_type{};
    auto returnObject = p.get_return_object();
    co_await p.initial_suspend();
    try {
        co_await boilWaterAsync(); //suspend & resume
        co_await makeTeaAsync();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





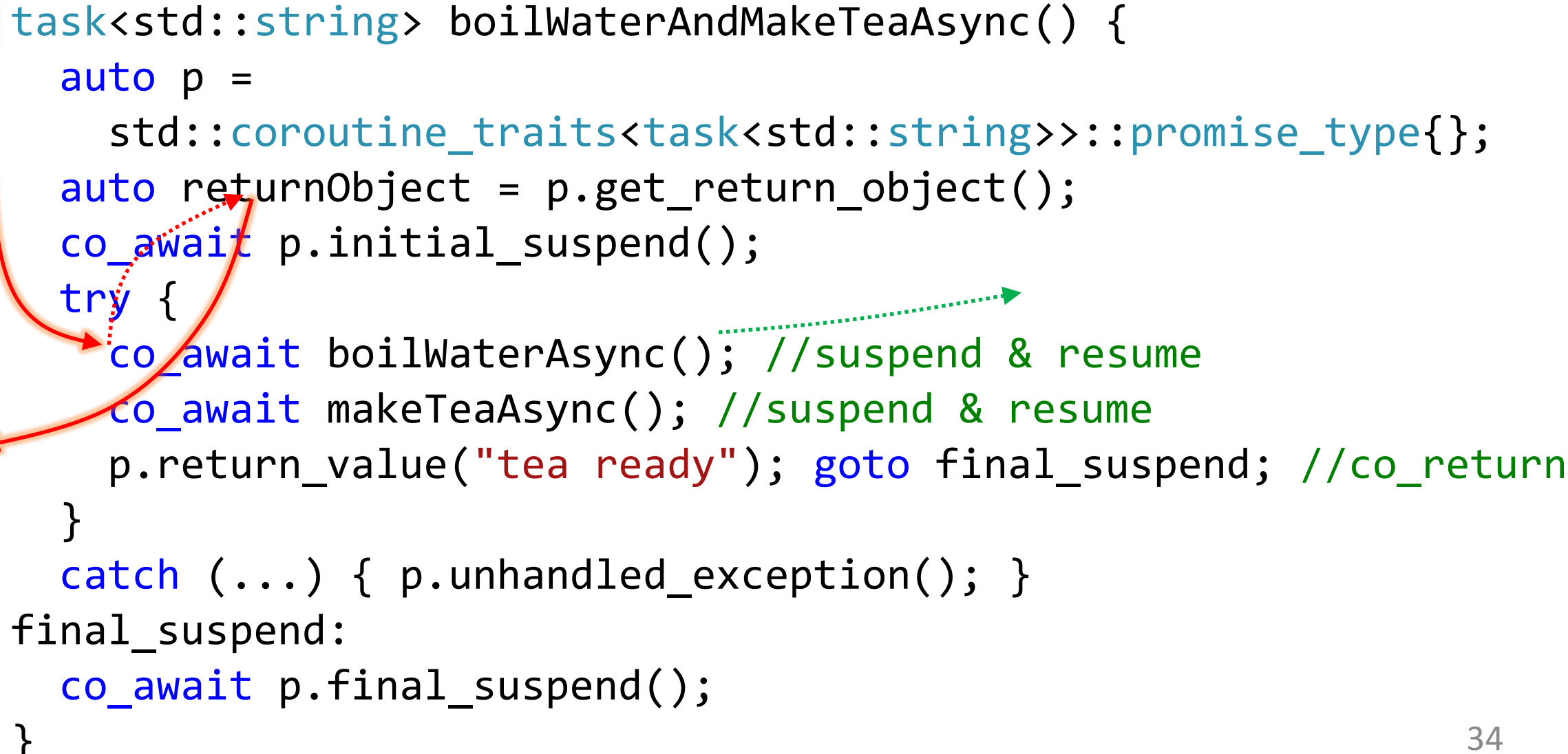
# Что происходит "за кулисами"

```
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();
}
```



# Что происходит "за кулисами"

```
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();  
}
```



# Что происходит "за кулисами"

```
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();  
}
```

# Что происходит "за кулисами"

```
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.

# Что происходит "за кулисами"

```
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();  
}
```

Что происходит "за кулисами"

```
co_yield expression;
```



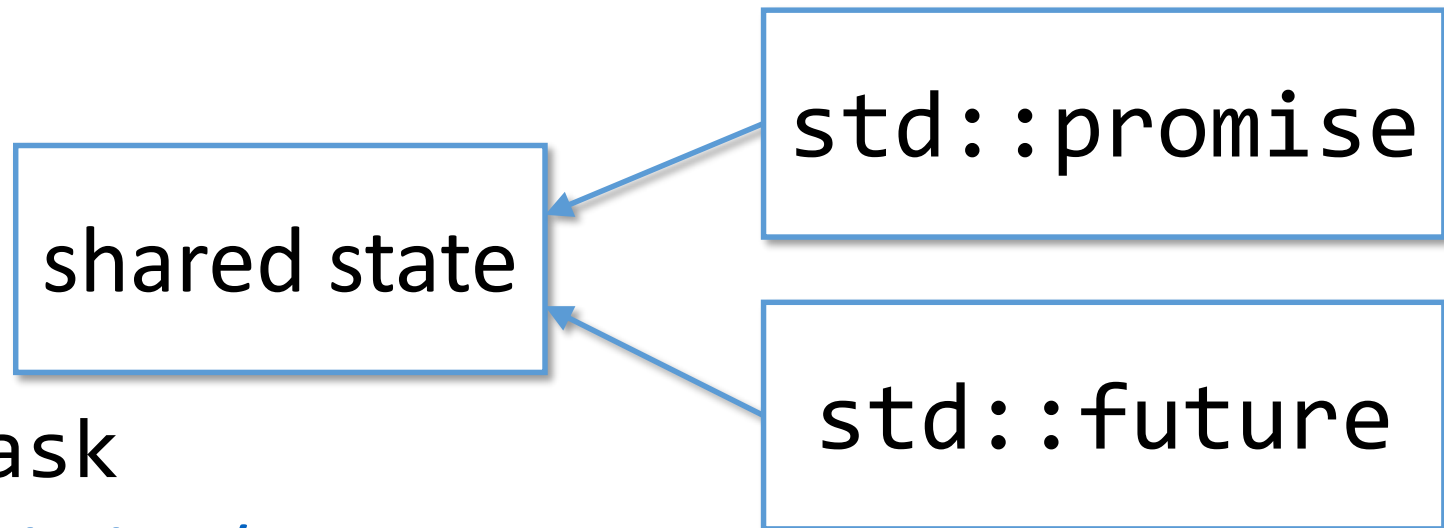
```
co_await promise.yield_value(expression);
```

Gotta go faster



# `std::future` неэффективен

- выделение памяти для разделяемого состояния
- синхронизация (ref count, get/set)
- накладные расходы на планирование продолжений



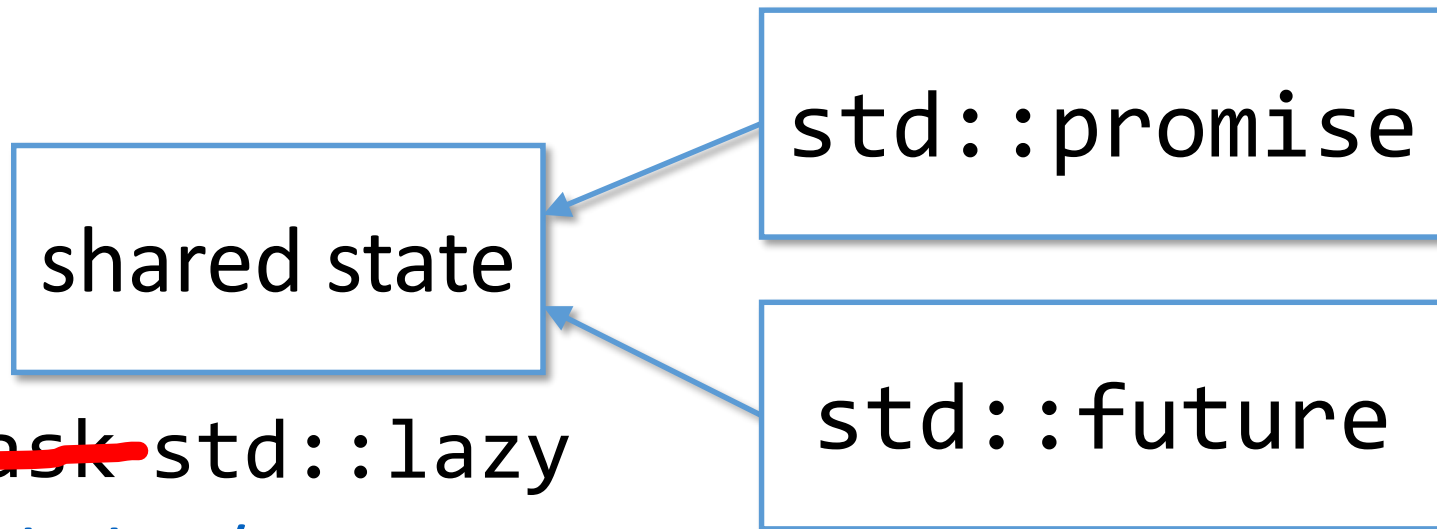
предложение [P1056](#)  
в стандарт C++: `std::task`

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



# `std::future` неэффективен

- выделение памяти для разделяемого состояния
- синхронизация (ref count, get/set)
- накладные расходы на планирование продолжений



предложение [P1056](#)

в стандарт C++: ~~`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);
```

время=541 нс  
скорость=1x

# Benchmark: concurrency::task

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

время=7195 нс  
скорость=0,08x

# Benchmark: легковесный 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();  
}
```

время=204 нс  
скорость=2,7x

```
BENCHMARK(coroutines);
```

# 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

---

# легковесный 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;
};
```

# легковесный 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) };
}
```

Спасибо!





# Асинхронная разработка на C++

Павел Новиков

[pnovikov@aligntech.com](mailto:pnovikov@aligntech.com)

R&D Align Technology

The logo for Align Technology, featuring the word "align" in a bold, lowercase, sans-serif font. The letter "i" has a small blue dot above it.

Вопросы?