# Asynchronous C++ programming

Pavel Novikov

 @cpp_ape

Align Technology R&D

# 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();
```

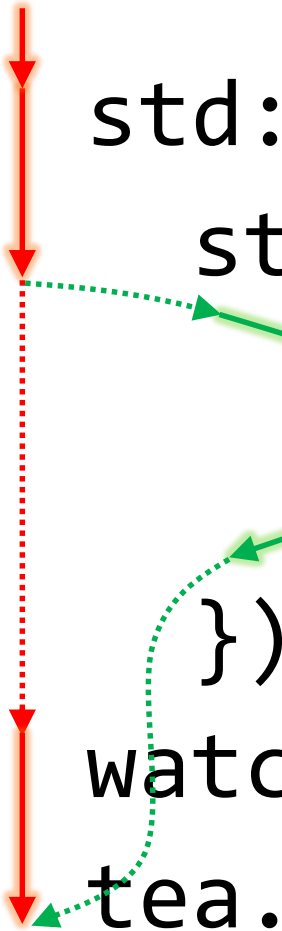
# std::async

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

# std::async

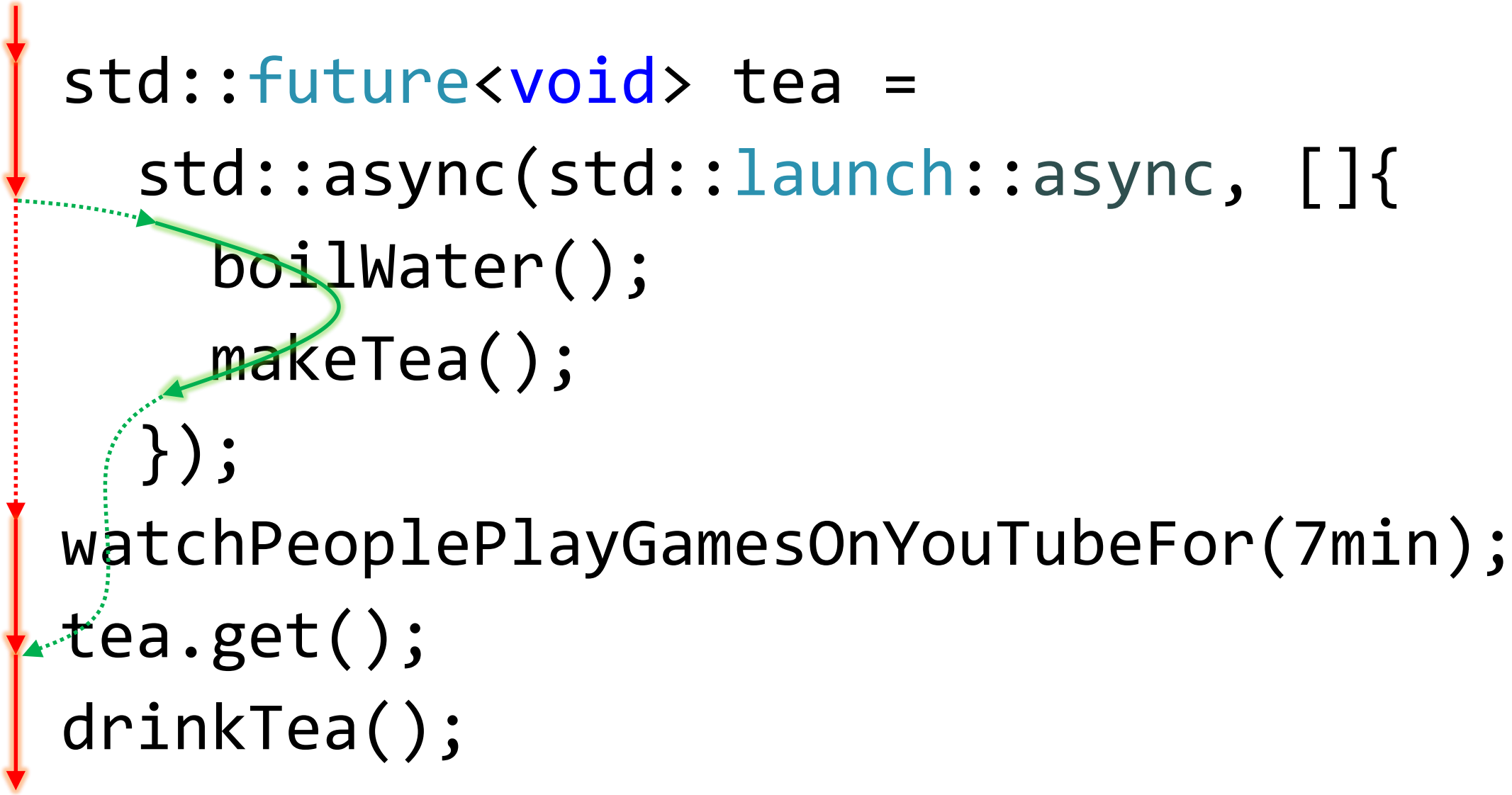
```
std::future<void> tea =
   std::async(std::launch::async, []{
      boilWater();
      makeTea();
   });
watchPeoplePlayGamesOnYouTubeFor(7min);
tea.get();
drinkTea();
```

# 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

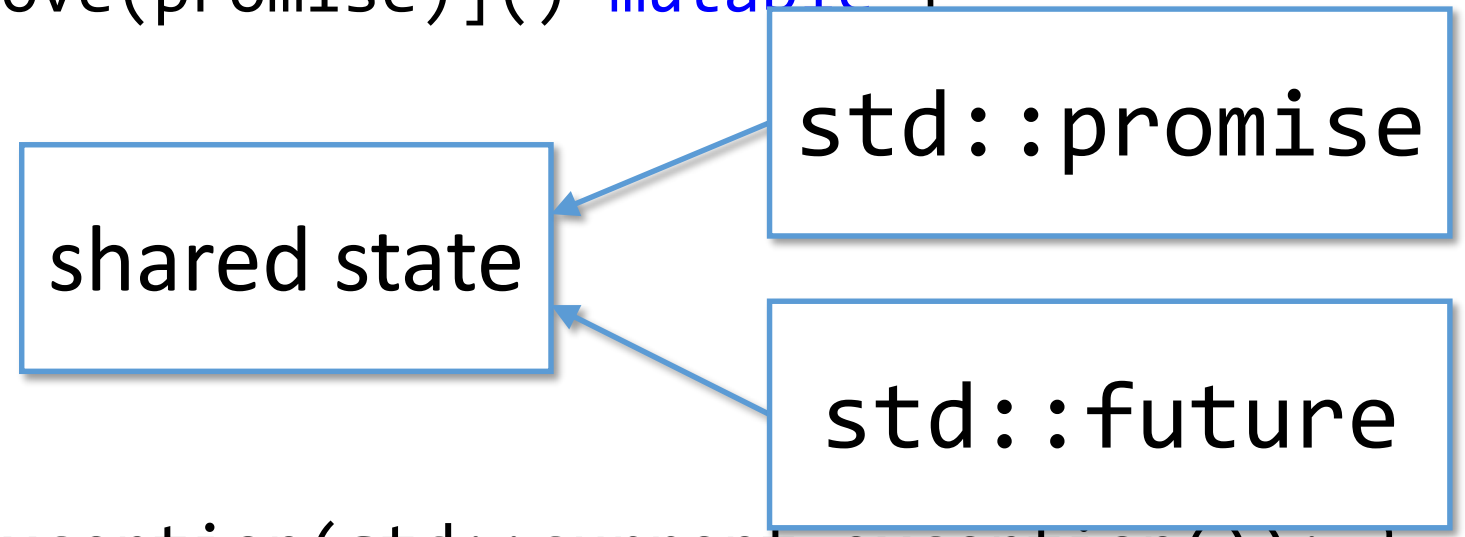
```
std::packaged_task<void()> task{ [](){
  boilWater();
  makeTea();
} };
std::future<void> tea = task.get_future();
help.execute(std::move(task));
tea.get();
drinkTea();
```

# 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();
```

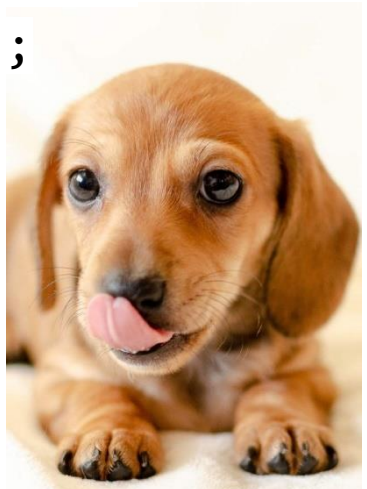
# std::promise

```
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();
```



# 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

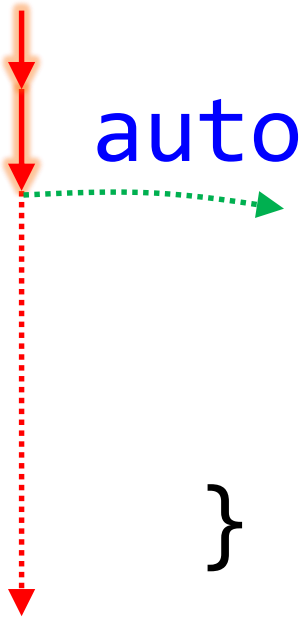
```
task<void> makeTeaAsync();

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();
```

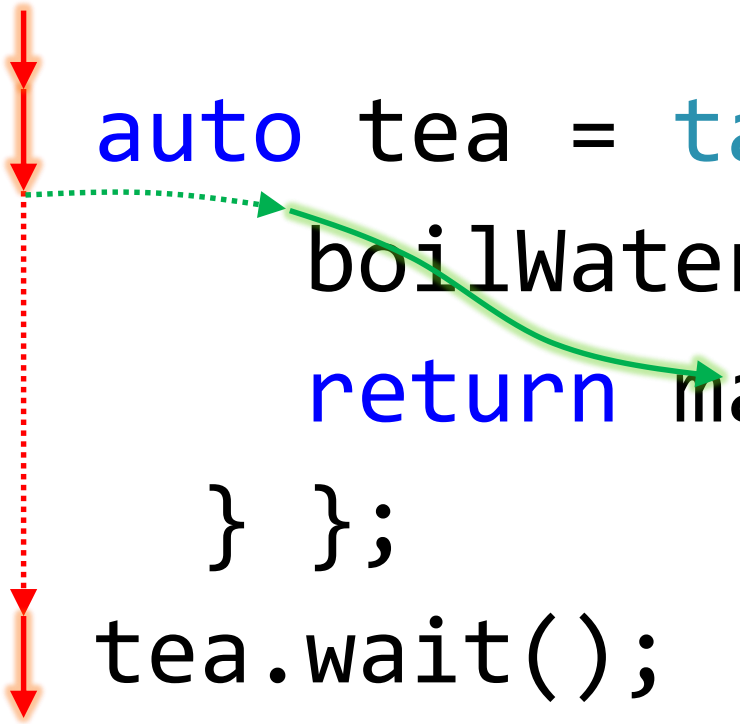
# Task unwrapping

```
task<void> makeTeaAsync();

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



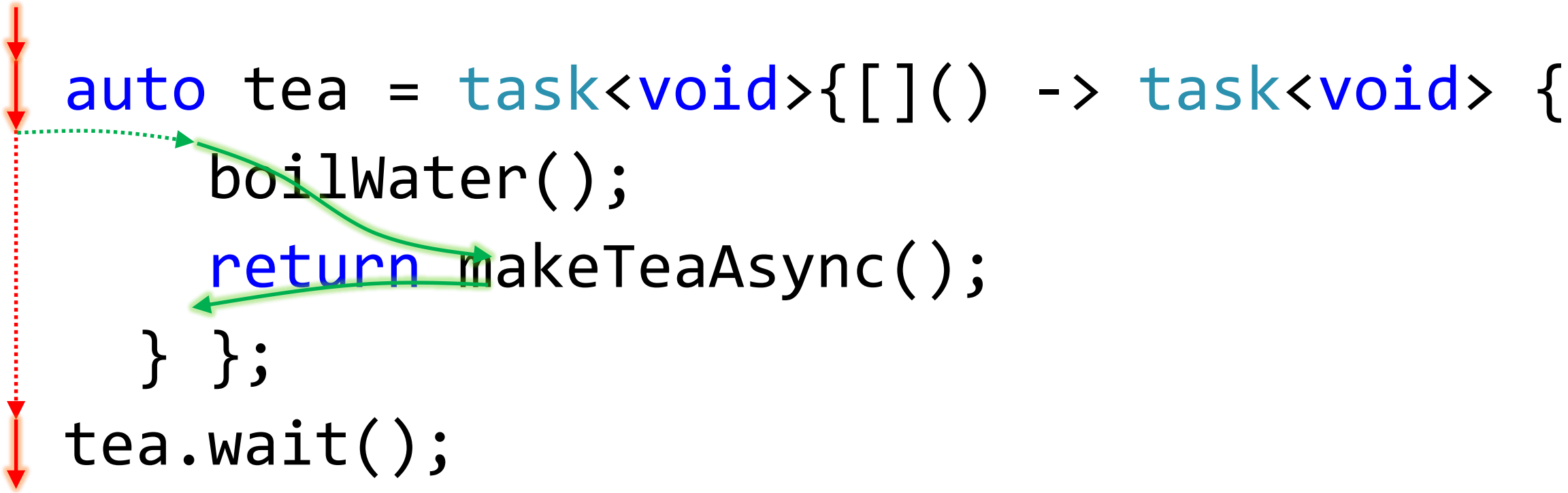
# Task unwrapping

```
task<void> makeTeaAsync();

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



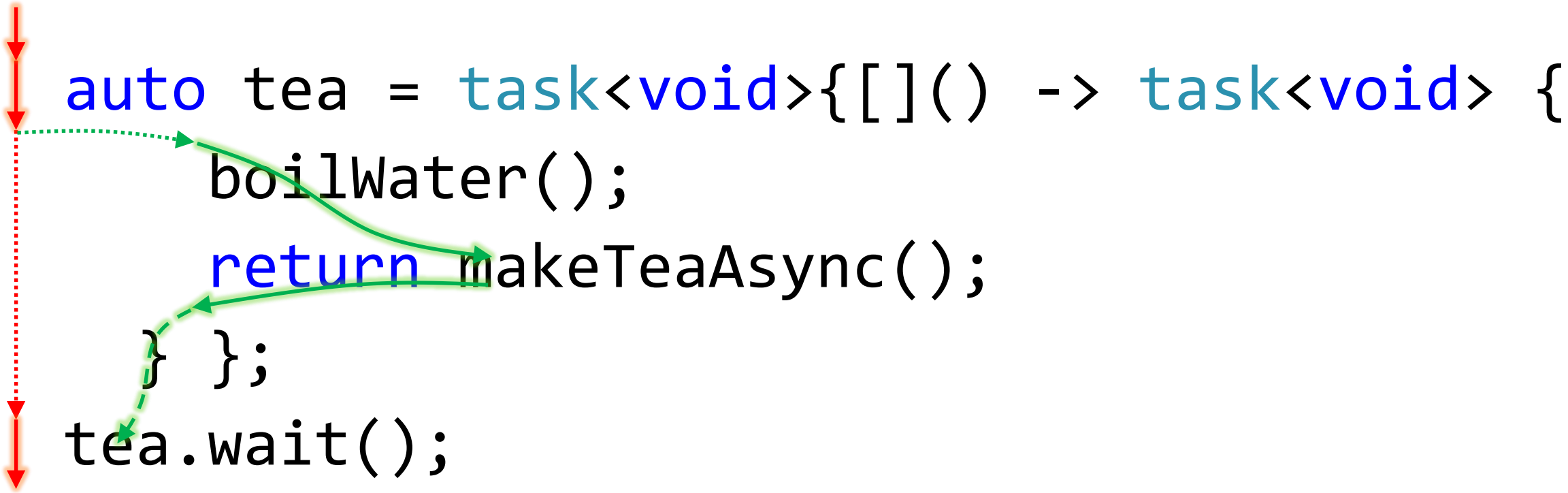
# Task unwrapping

```
task<void> makeTeaAsync();

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



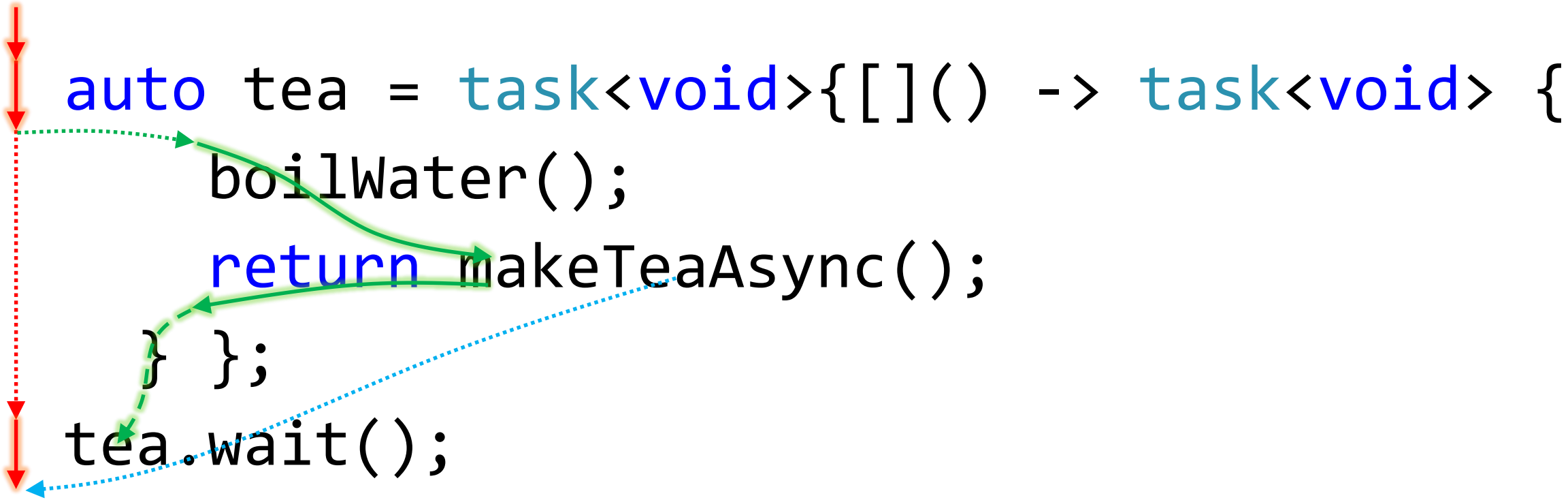
# Task unwrapping

```
task<void> makeTeaAsync();

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



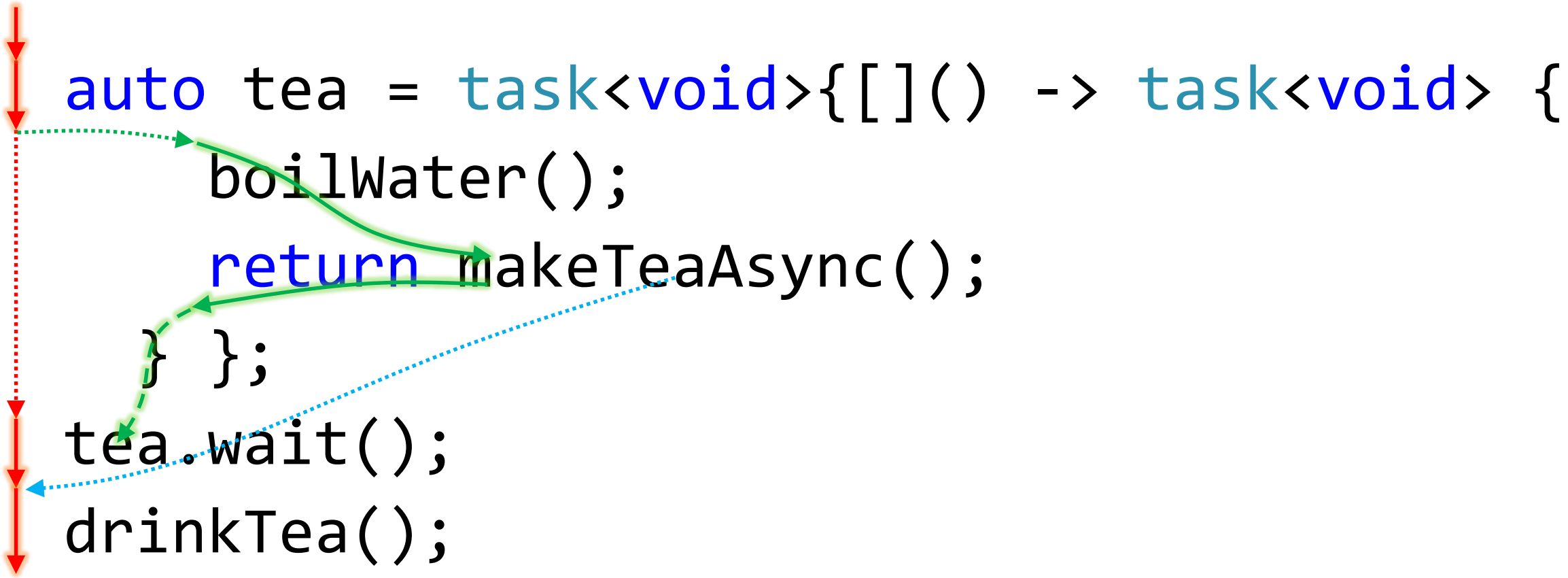
# Task unwrapping

```
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();
```



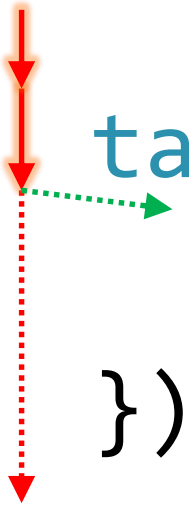
# Continuations

```
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

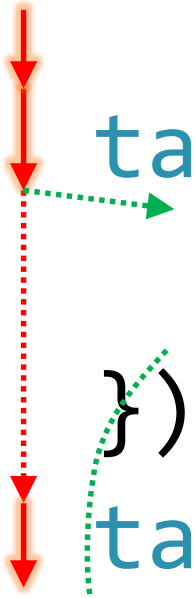


```
task<std::string> water = create_task([] {
  boilWater(); return "Water boiled"s;
});
task<void> tea = water.then(
  [](const std::string &msg) {
    makeTea();
  });
tea.wait();
```

# Continuations

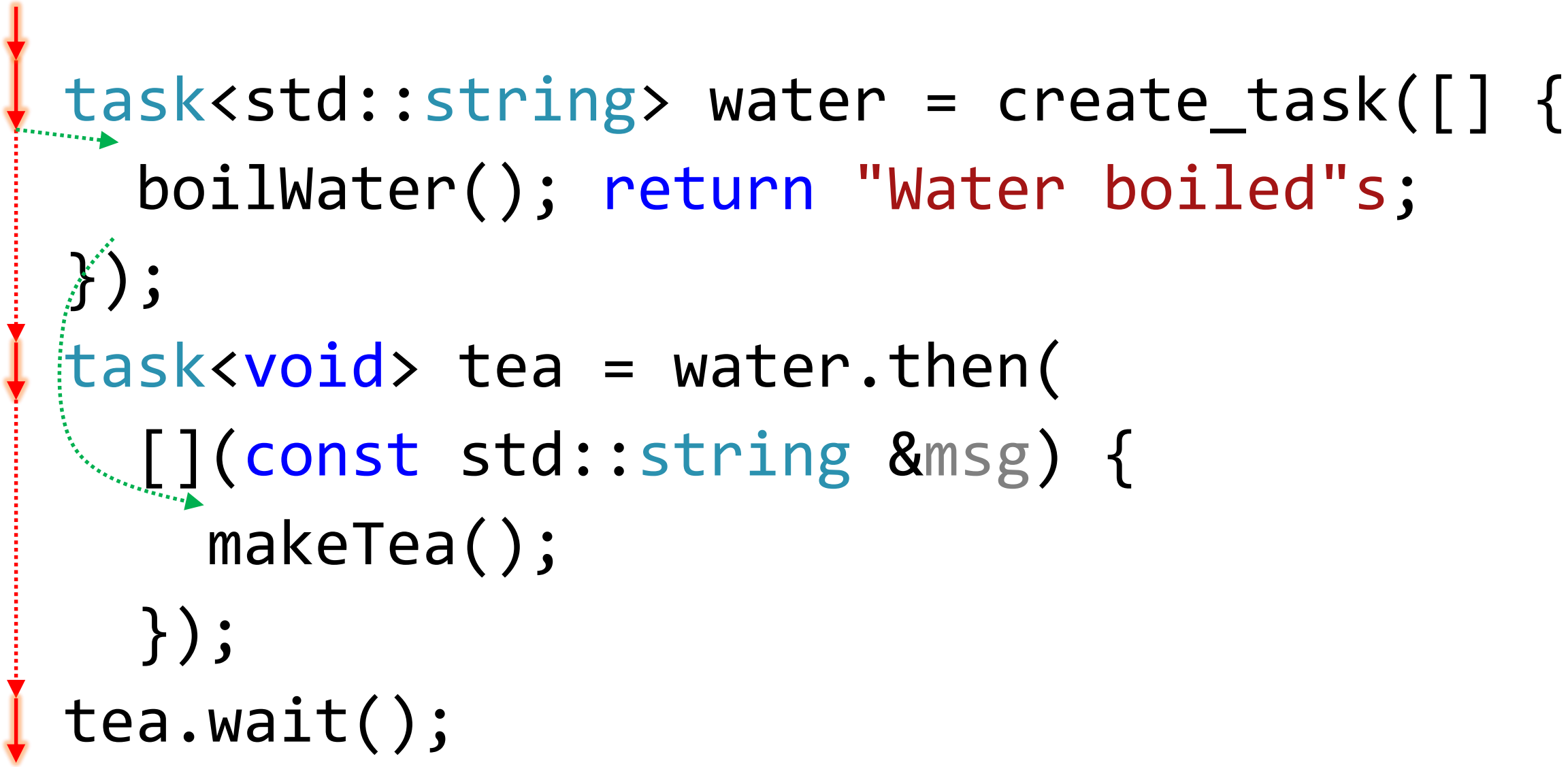
```
task<std::string> water = create_task([] {
  boilWater(); return "Water boiled"s;
});
task<void> tea = water.then(
  [](const std::string &msg) {
    makeTea();
  });
tea.wait();
```

# Continuations



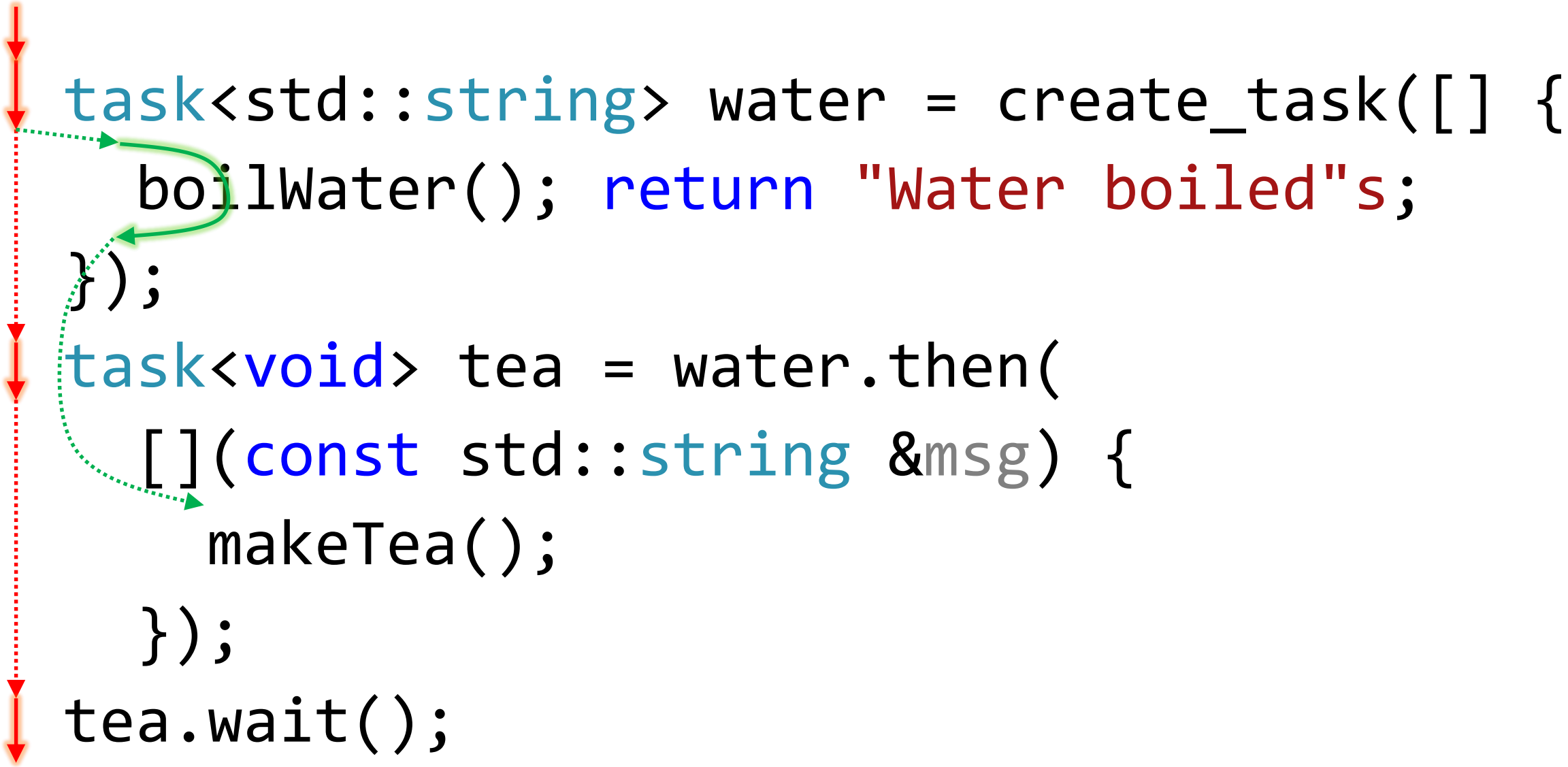
```
task<std::string> water = create_task([] {
    boilWater(); return "Water boiled"s;
});
task<void> tea = water.then(
    [](const std::string &msg) {
        makeTea();
    });
tea.wait();
```

# Continuations



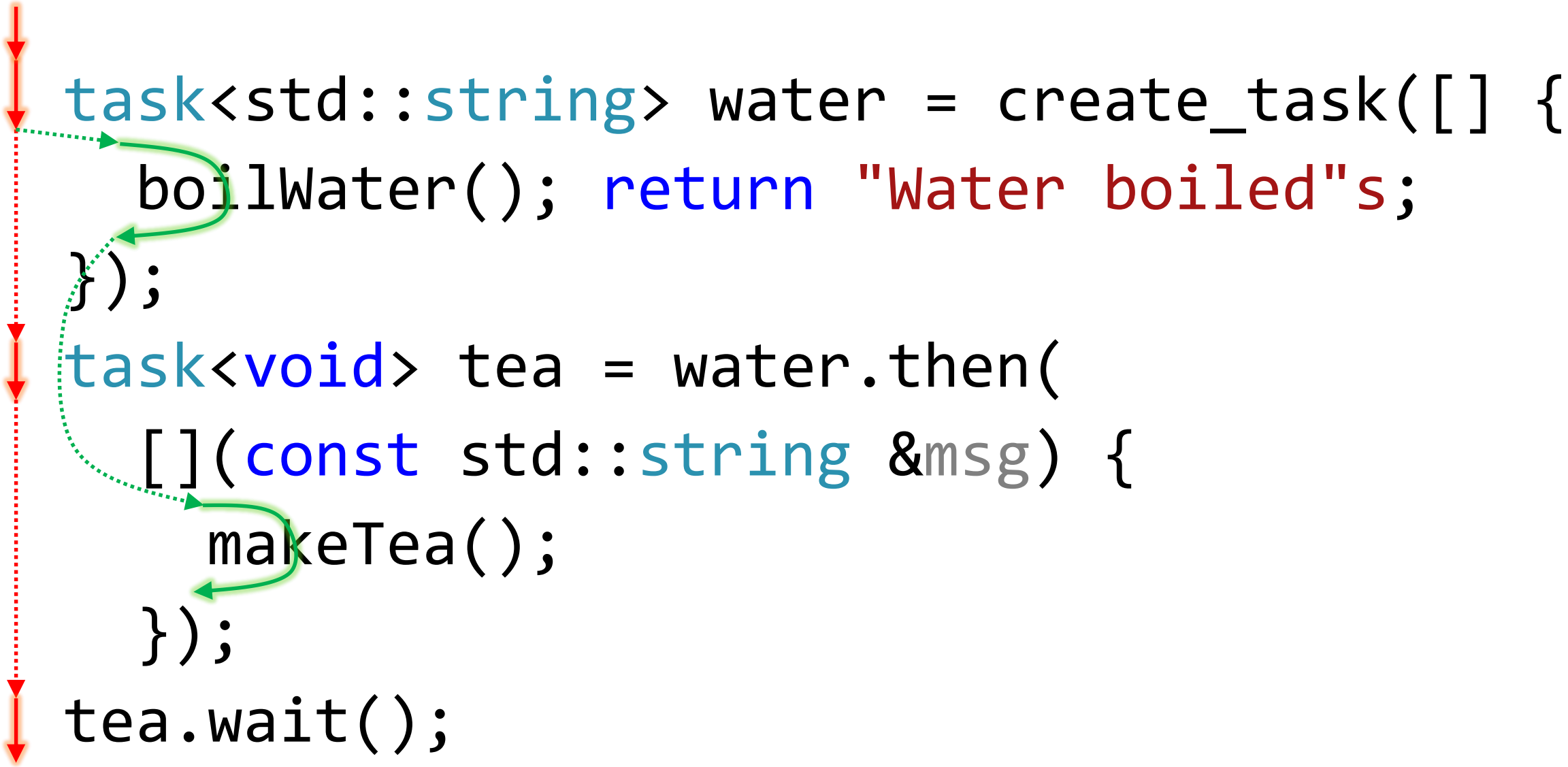
```
task<std::string> water = create_task([] {
    boilWater(); return "Water boiled"s;
});
task<void> tea = water.then(
    [](const std::string &msg) {
        makeTea();
    });
tea.wait();
```

# Continuations



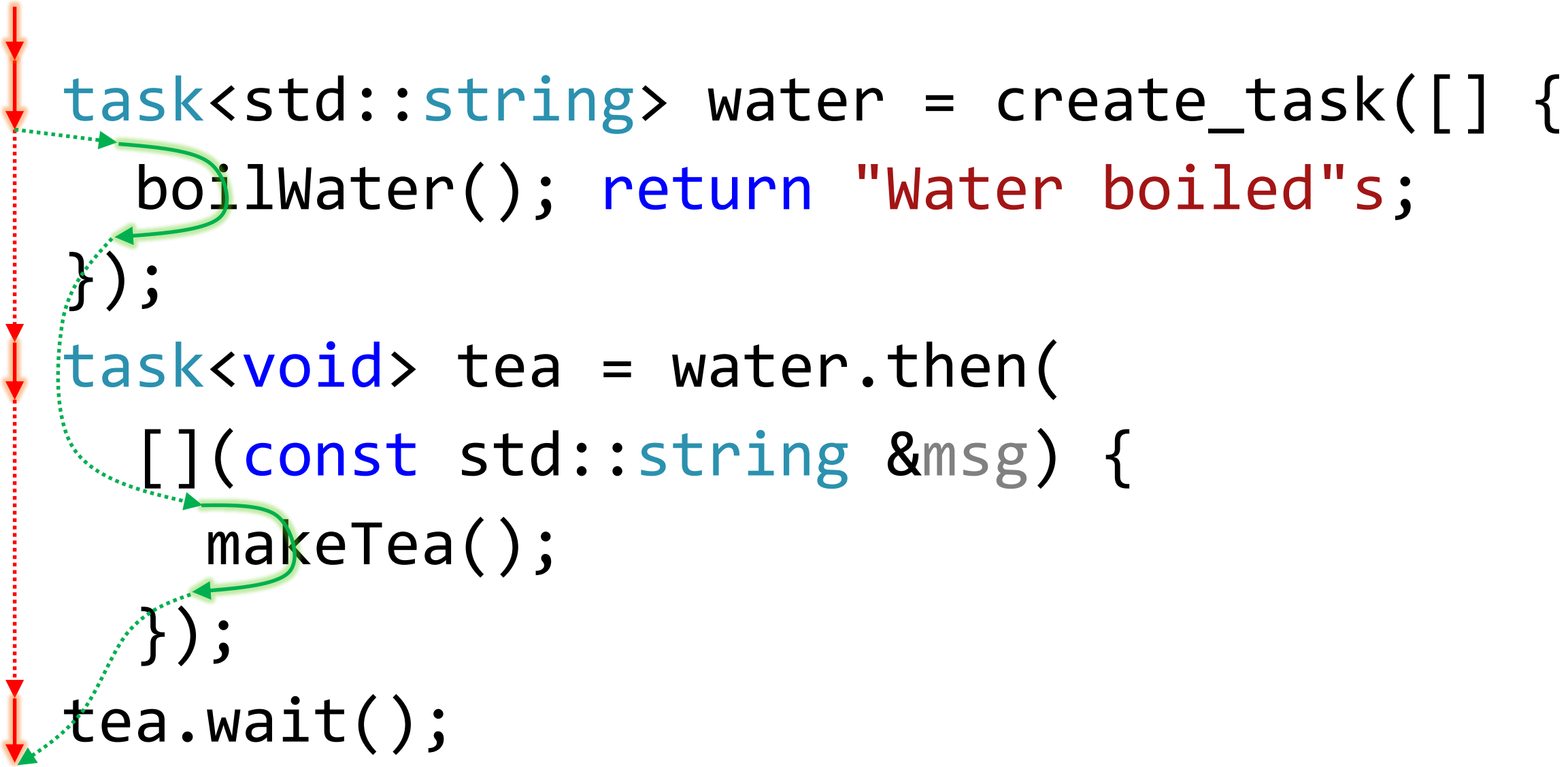
```
task<std::string> water = create_task([] {
    boilWater(); return "Water boiled"s;
});
task<void> tea = water.then(
    [](const std::string &msg) {
        makeTea();
    });
tea.wait();
```

# Continuations



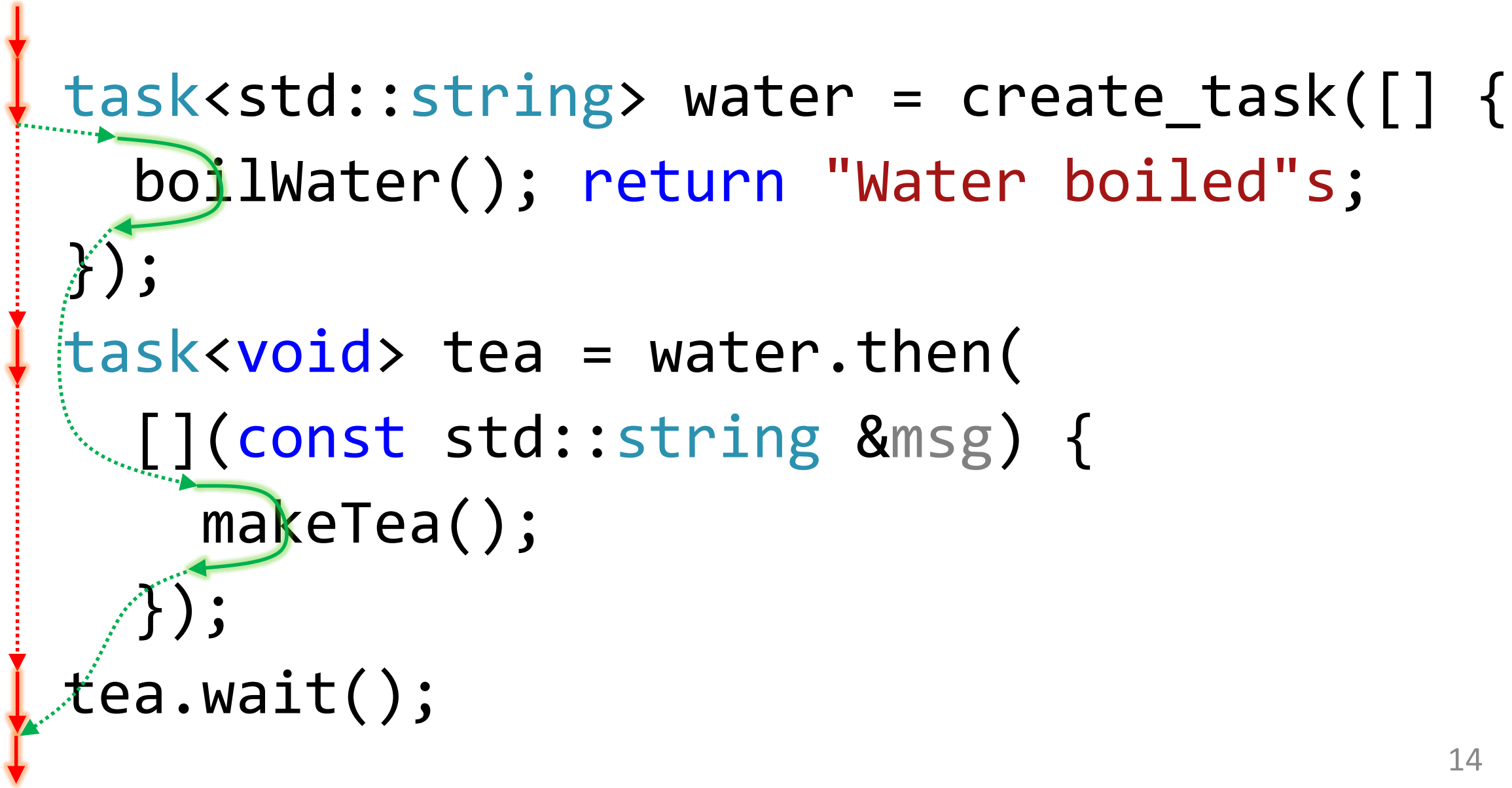
```
task<std::string> water = create_task([] {
    boilWater(); return "Water boiled"s;
});
task<void> tea = water.then(
    [](const std::string &msg) {
        makeTea();
    });
tea.wait();
```

# Continuations



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

# Continuations



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

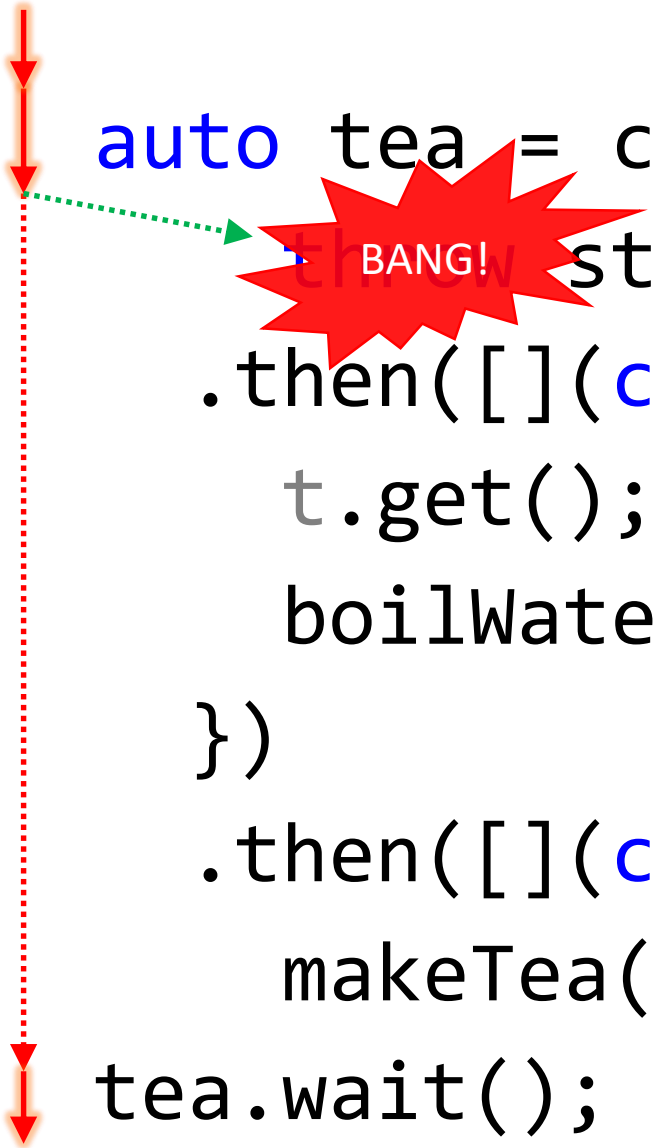
# Exception handling

```
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

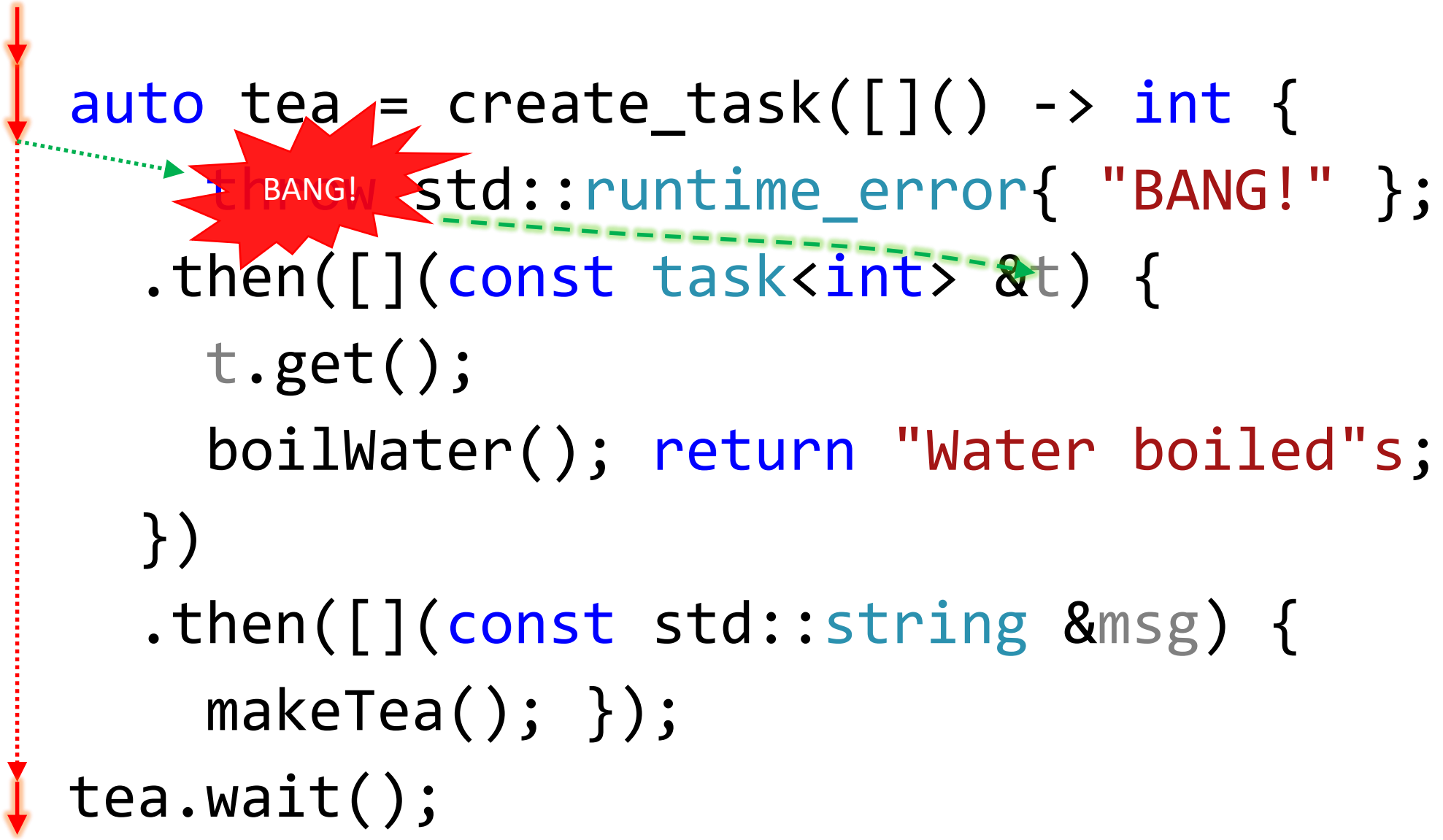
```
auto tea = create_task([]() -> int {
    throw std::runtime_error{ "BANG!" }; })
  .then([](const task<int> &t) {
    t.get();
    boilWater(); return "Water boiled"s;
  })
  .then([](const std::string &msg) {
    makeTea(); });
tea.wait();
```

# Exception handling



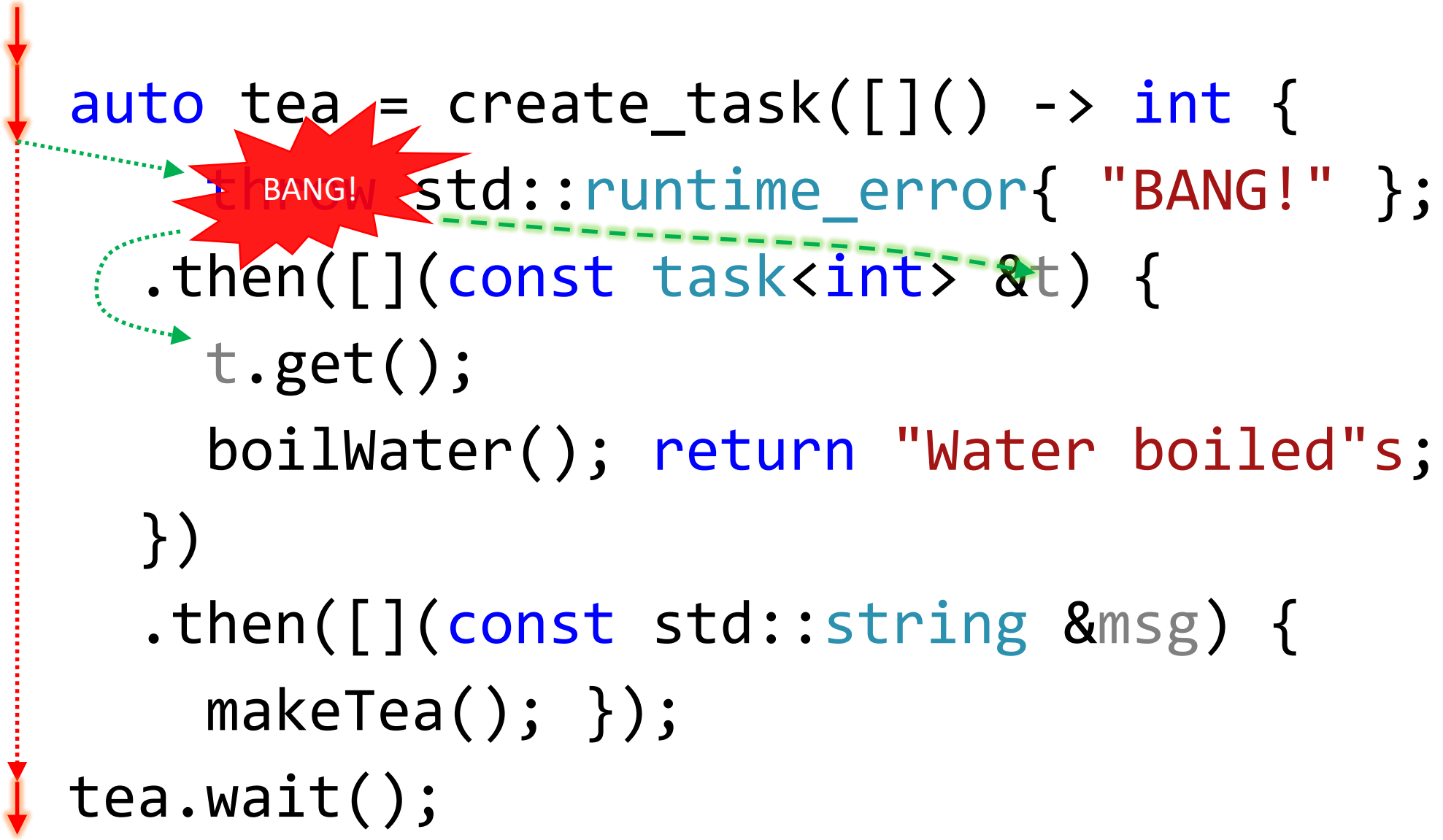

```
auto tea = create_task([]() -> int {
    throw std::runtime_error{ "BANG!" }; })
  .then([](const task<int> &t) {
    t.get();
    boilWater(); return "Water boiled"s;
  })
  .then([](const std::string &msg) {
    makeTea(); });
tea.wait();
```

# Exception handling



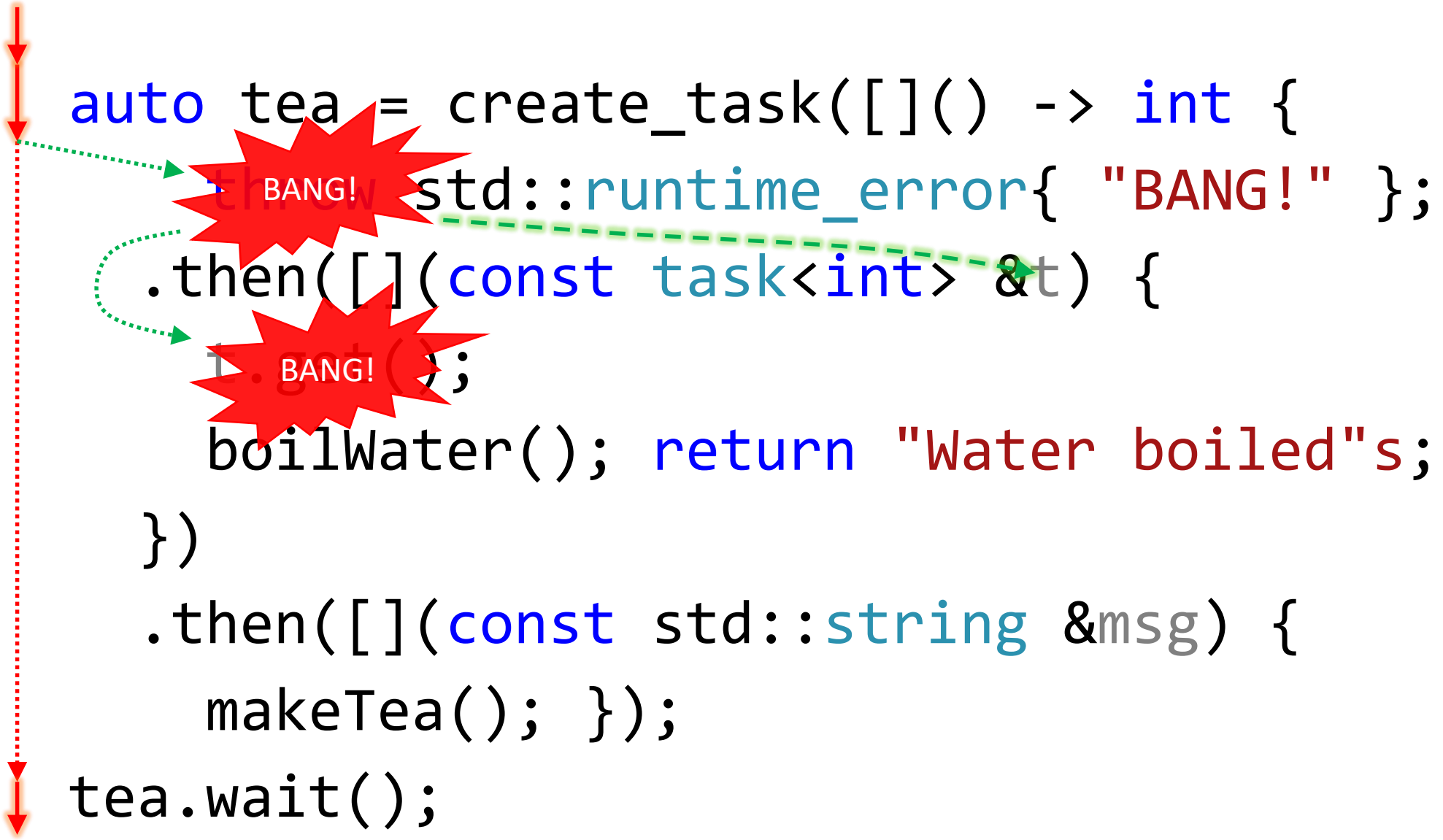

```
auto tea = create_task([]() -> int {
    throw std::runtime_error{ "BANG!" }; })
  .then([](const task<int> &t) {
    t.get();
    boilWater(); return "Water boiled"s;
  })
  .then([](const std::string &msg) {
    makeTea(); });
tea.wait();
```

# Exception handling

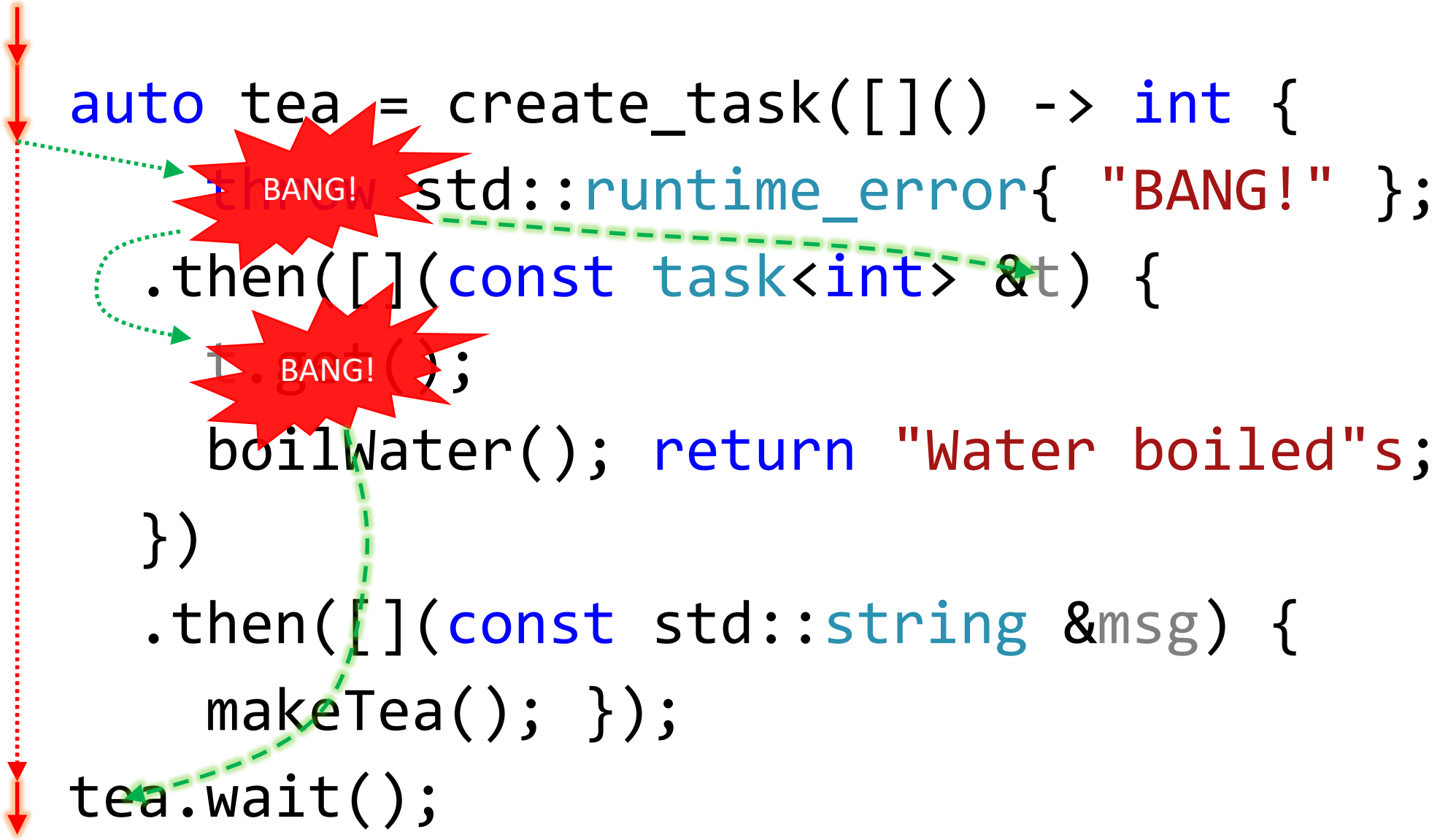



```
auto tea = create_task([]() -> int {
    throw std::runtime_error{ "BANG!" }; })
  .then([](const task<int> &t) {
    t.get();
    boilWater(); return "Water boiled"s;
  })
  .then([](const std::string &msg) {
    makeTea(); });
tea.wait();
```

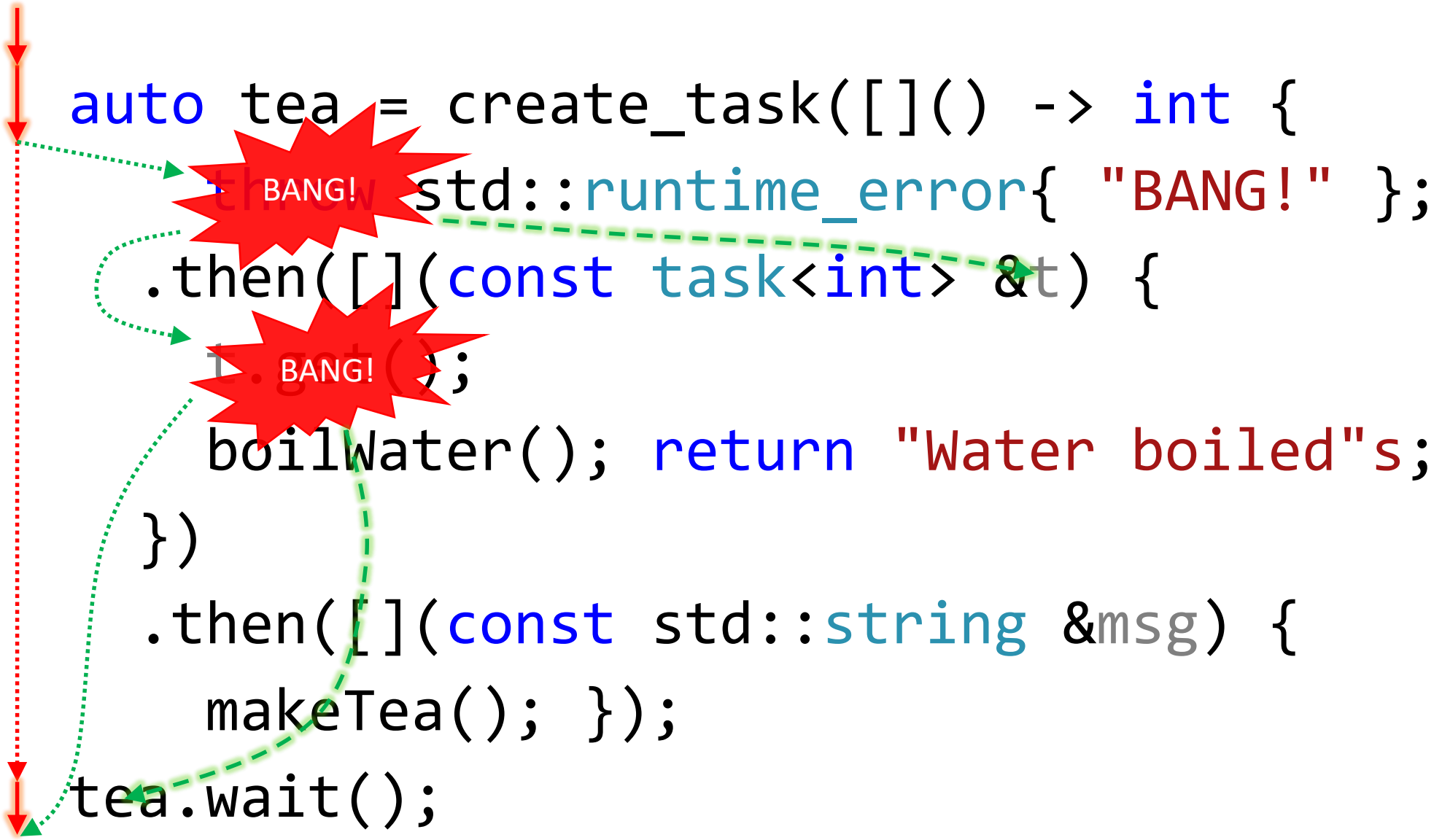
# Exception handling



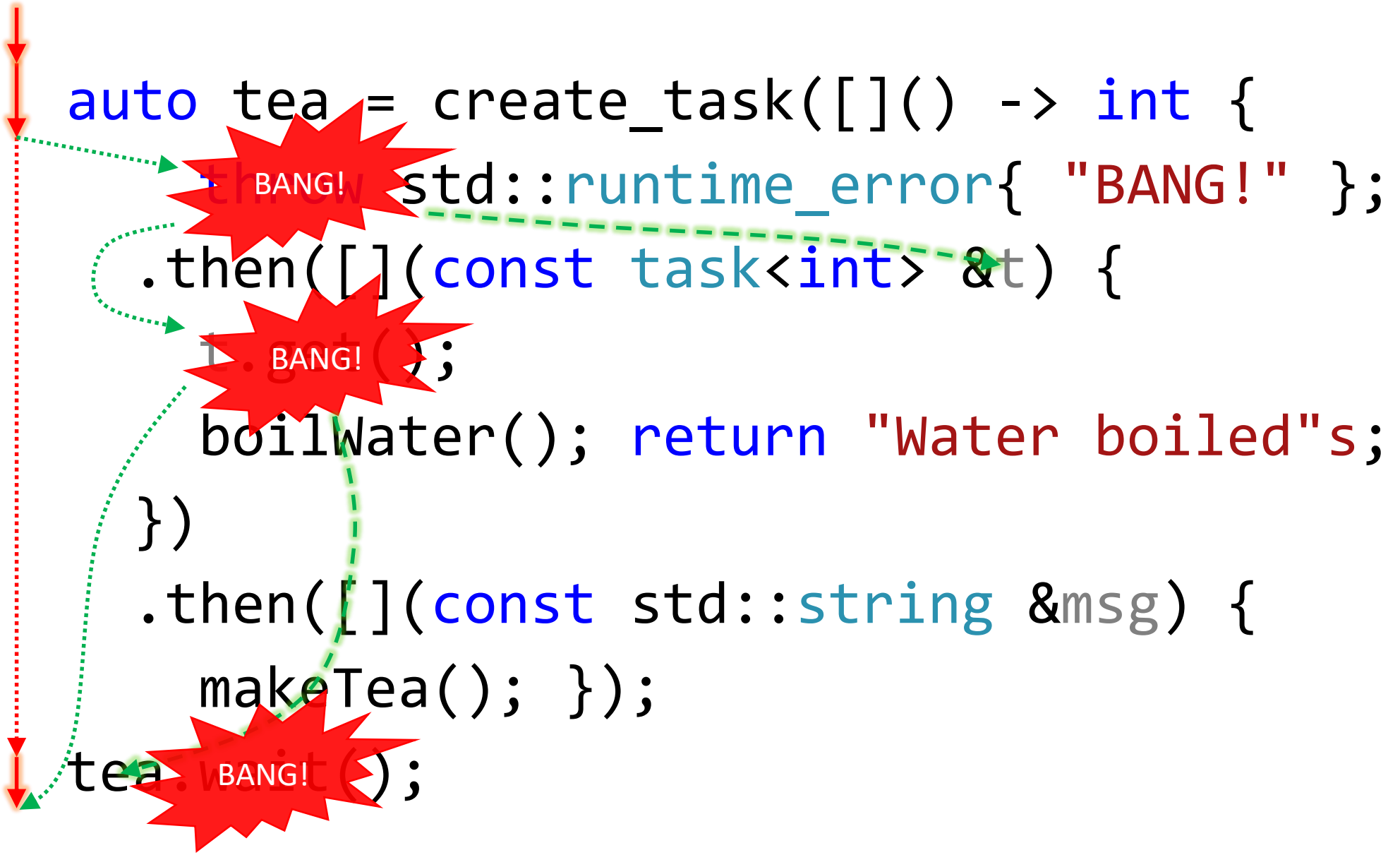
# Exception handling



# Exception handling



# Exception handling



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

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

returns task result and its index

# Idiom: do while

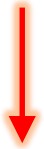
```
template<typename Func>
task<void> doWhile(Func func) {
  return create_task(func)
    .then([func](bool needToContinue) {
      if (needToContinue)
        return doWhile(func);
      return task_from_result();
    });
}
```

# 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();
    });
}
```

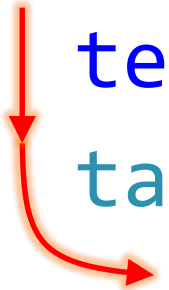
a sea puppy

# Idiom: do while



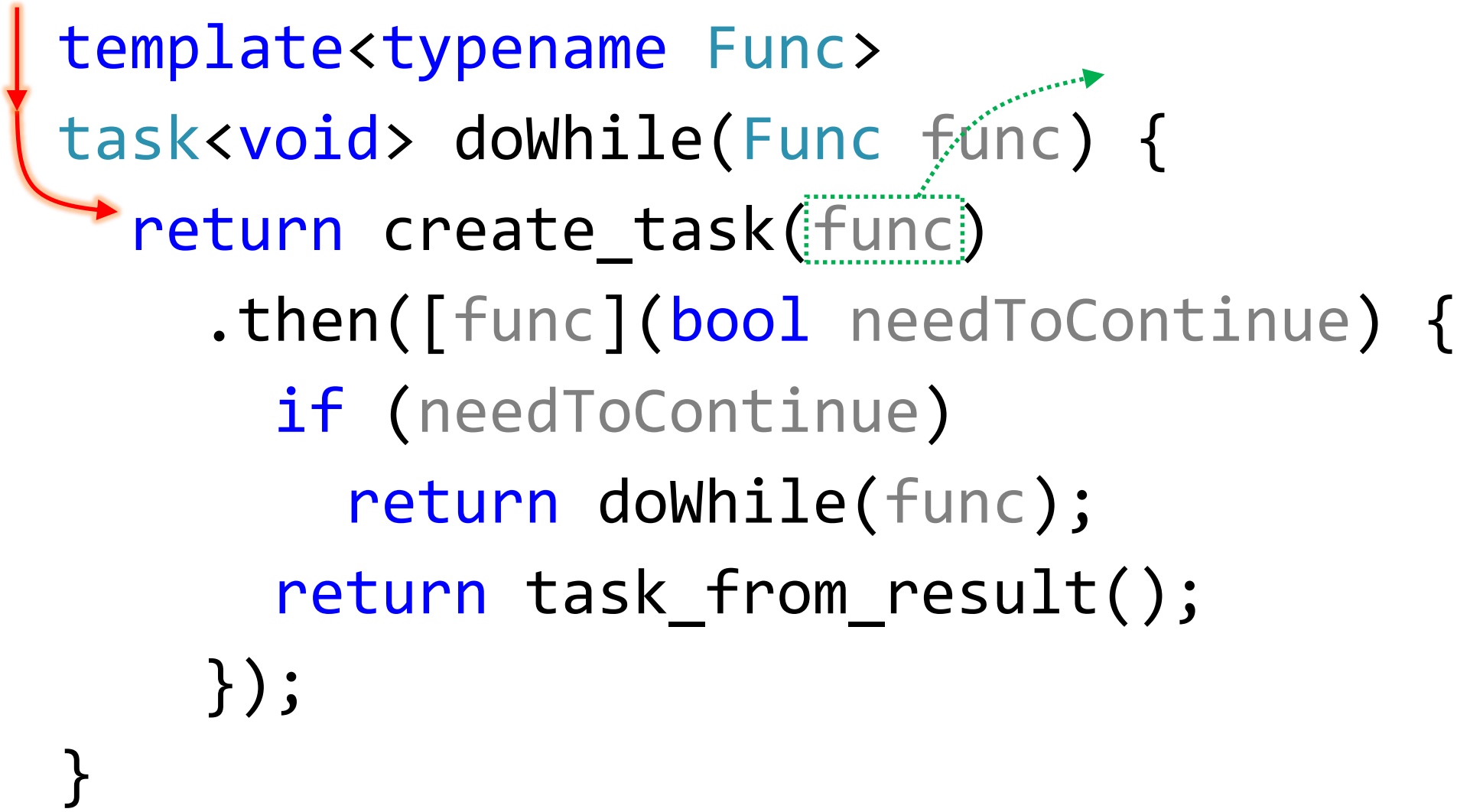
```
template<typename Func>
task<void> doWhile(Func func) {
  return create_task(func)
    .then([func](bool needToContinue) {
      if (needToContinue)
        return doWhile(func);
      return task_from_result();
    });
}
```

# Idiom: do while



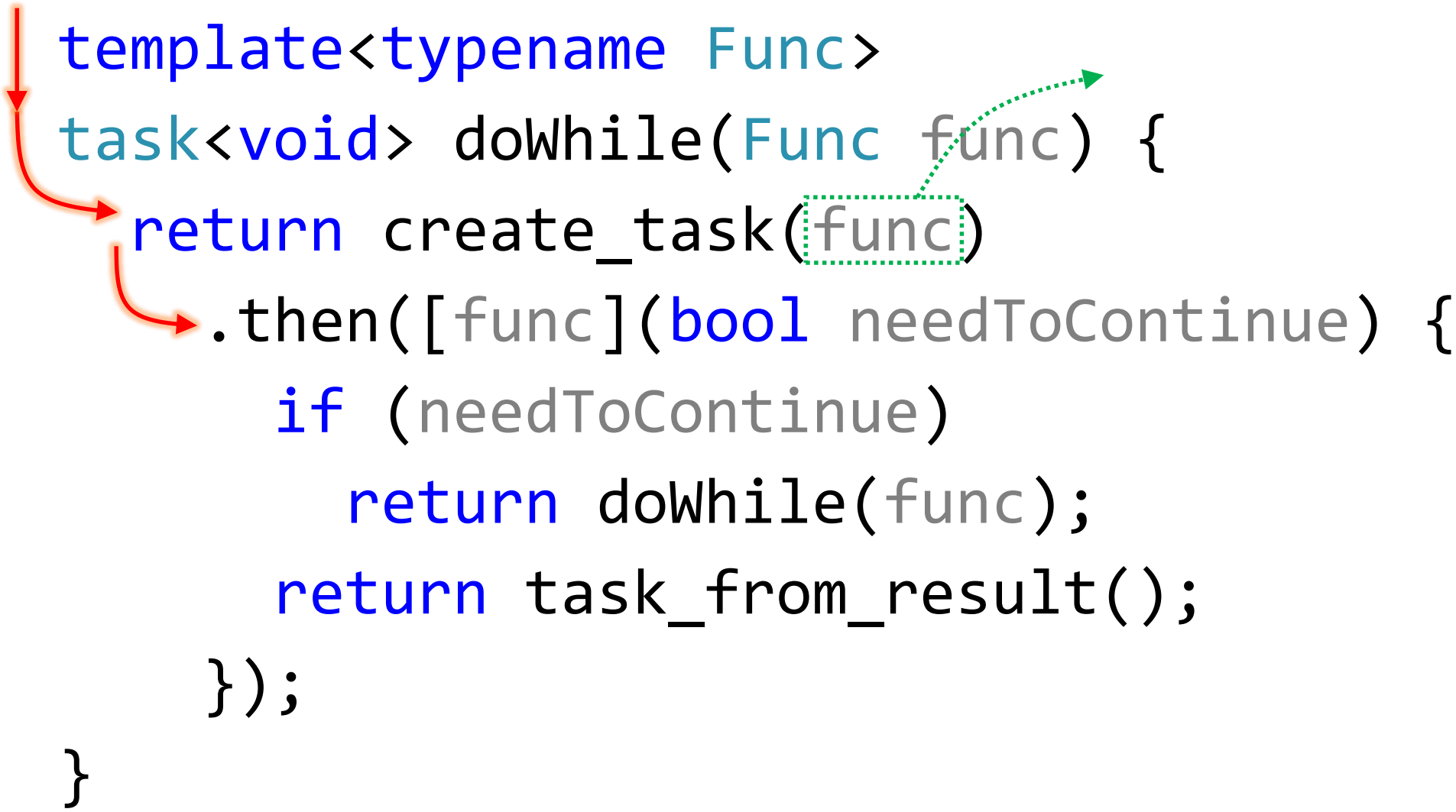
```
template<typename Func>
task<void> doWhile(Func func) {
  return create_task(func)
    .then([func](bool needToContinue) {
      if (needToContinue)
        return doWhile(func);
      return task_from_result();
    });
}
```

# 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();
    });
}
```

# Idiom: do while

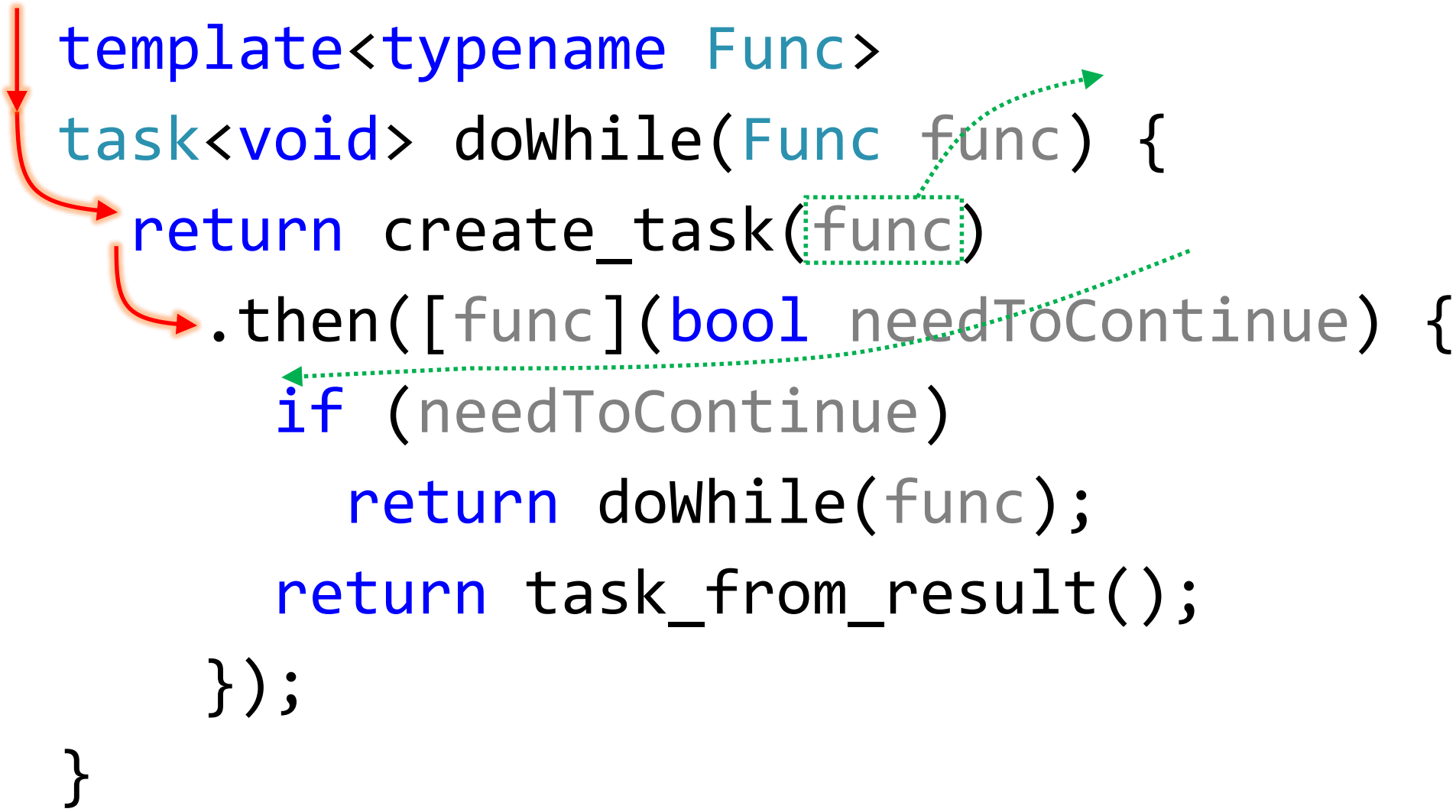


```
template<typename Func>
task<void> doWhile(Func func) {
  return create_task(func)
    .then([func](bool needToContinue) {
      if (needToContinue)
        return doWhile(func);
      return task_from_result();
    });
}
```

# Idiom: do while

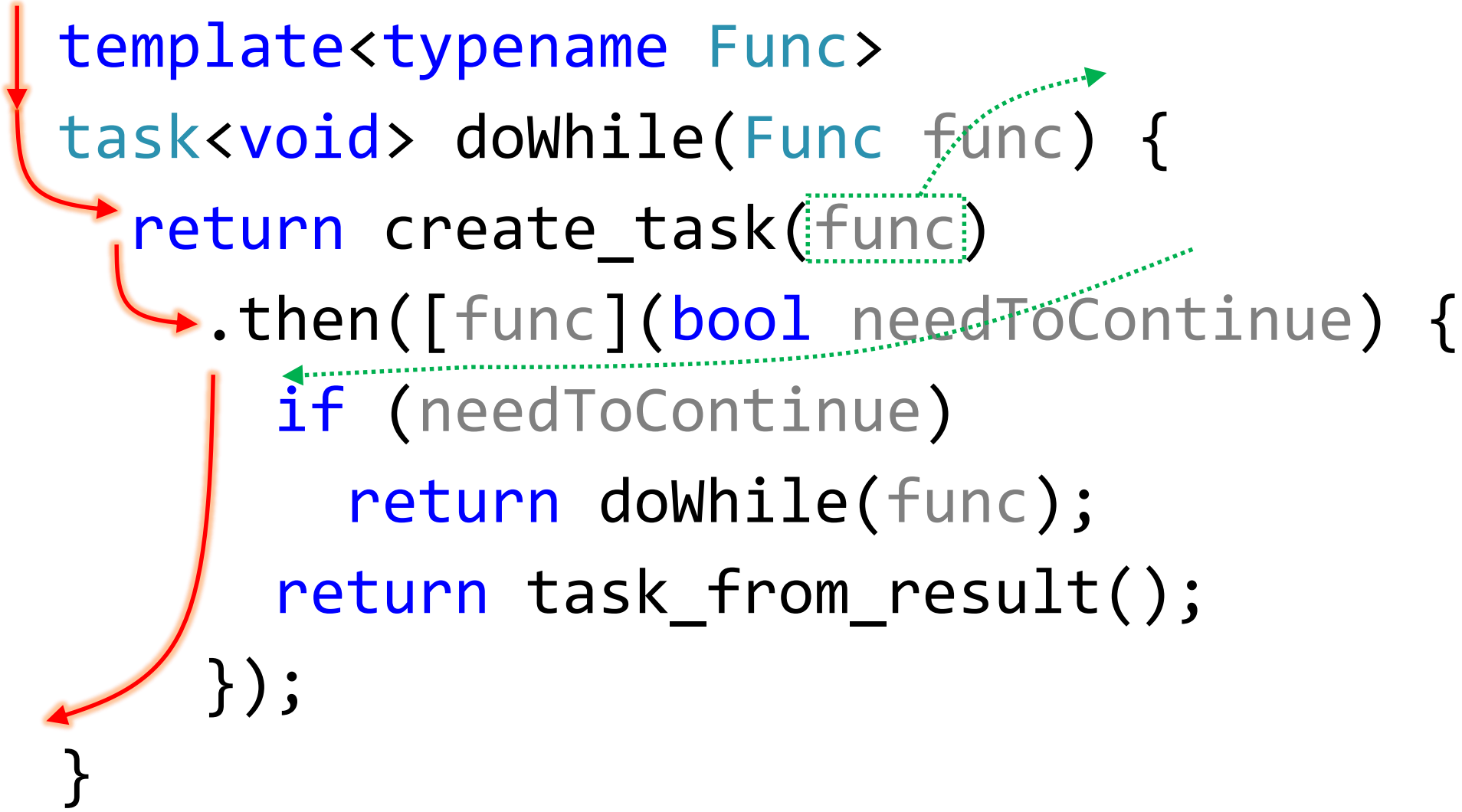
```
template<typename Func>
task<void> doWhile(Func func) {
  return create_task(func)
    .then([func](bool needToContinue) {
      if (needToContinue)
        return doWhile(func);
      return task_from_result();
    });
}
```

# Idiom: do while

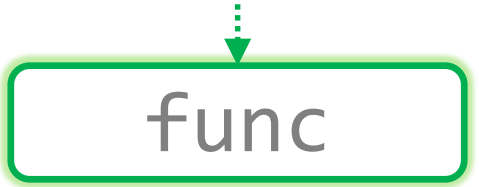


```
template<typename Func>
task<void> doWhile(Func func) {
  return create_task(func)
    .then([func](bool needToContinue) {
      if (needToContinue)
        return doWhile(func);
      return task_from_result();
    });
}
```

# Idiom: do while

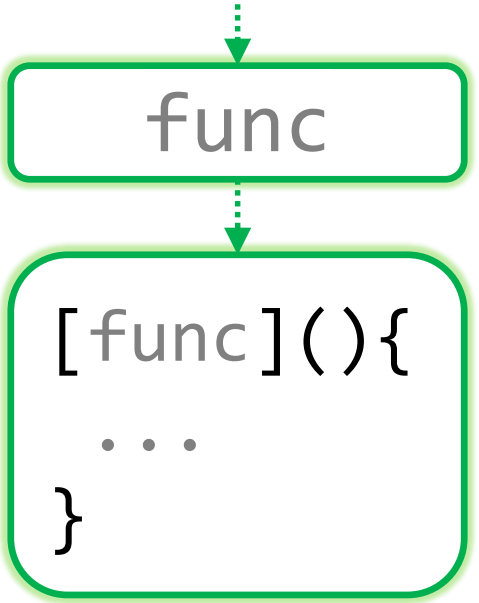


# Idiom: do while




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

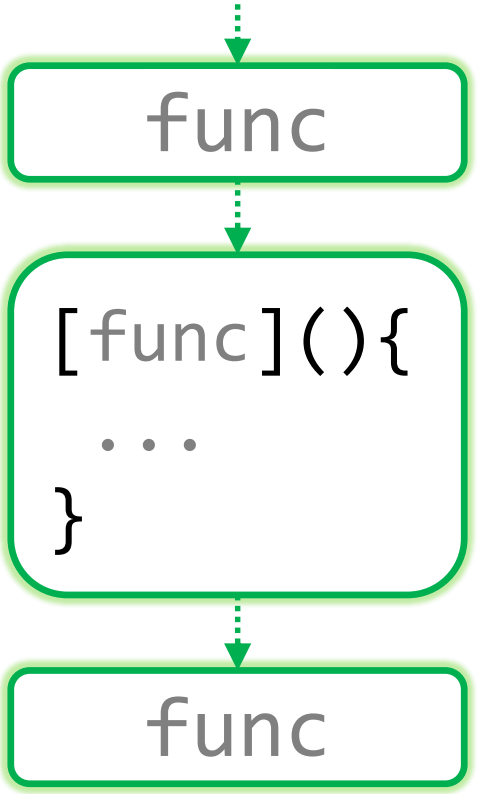
# Idiom: do while




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

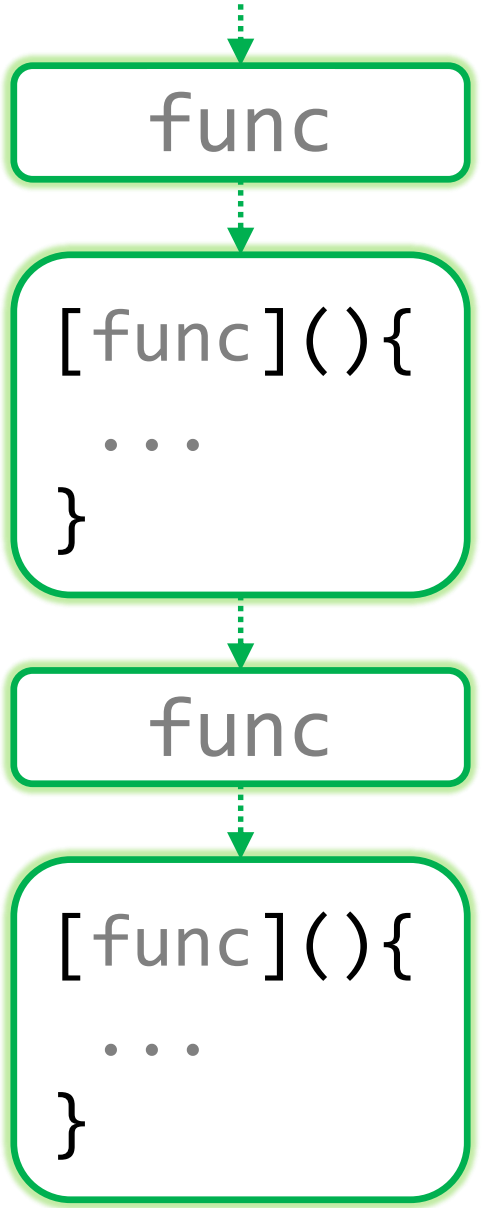
# Idiom: do while

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



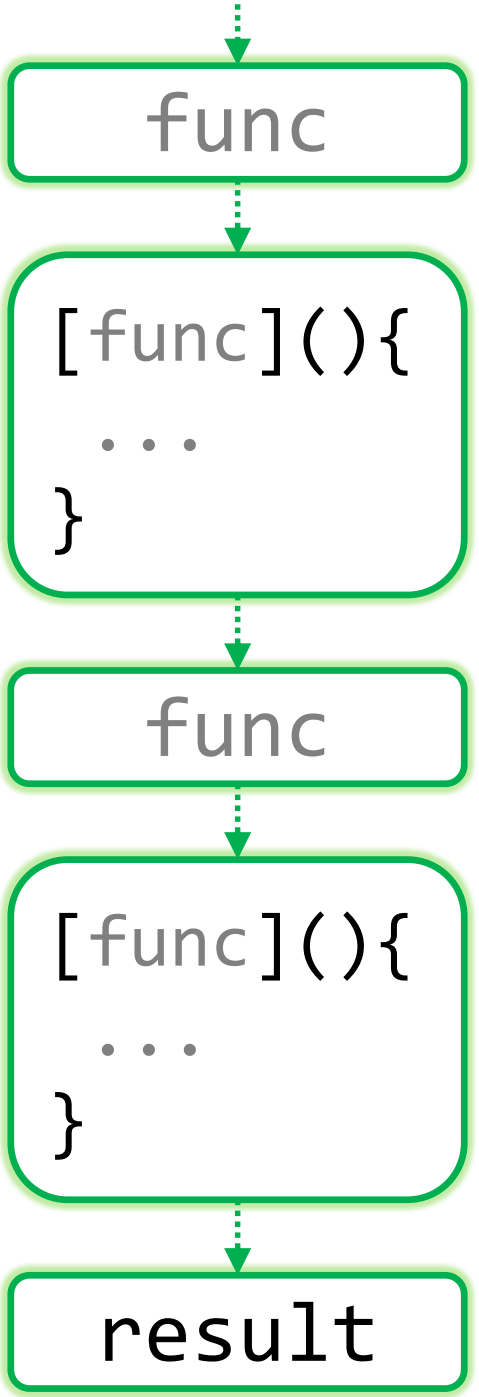
# Idiom: do while

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



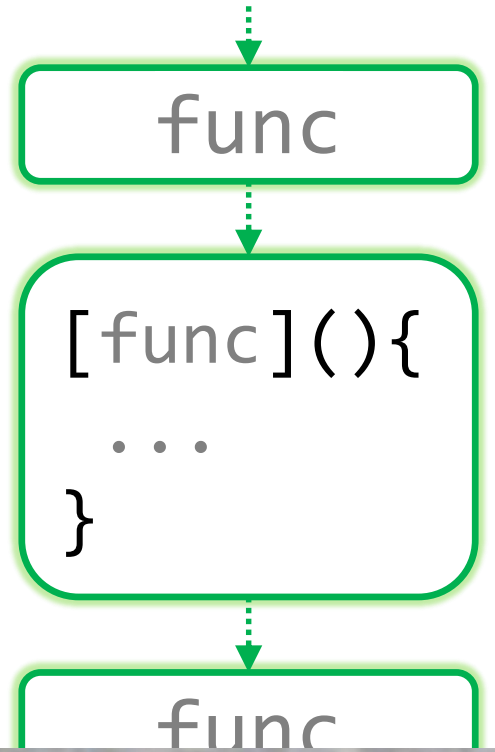
# Idiom: do while

```
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: 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();
    });
}
```

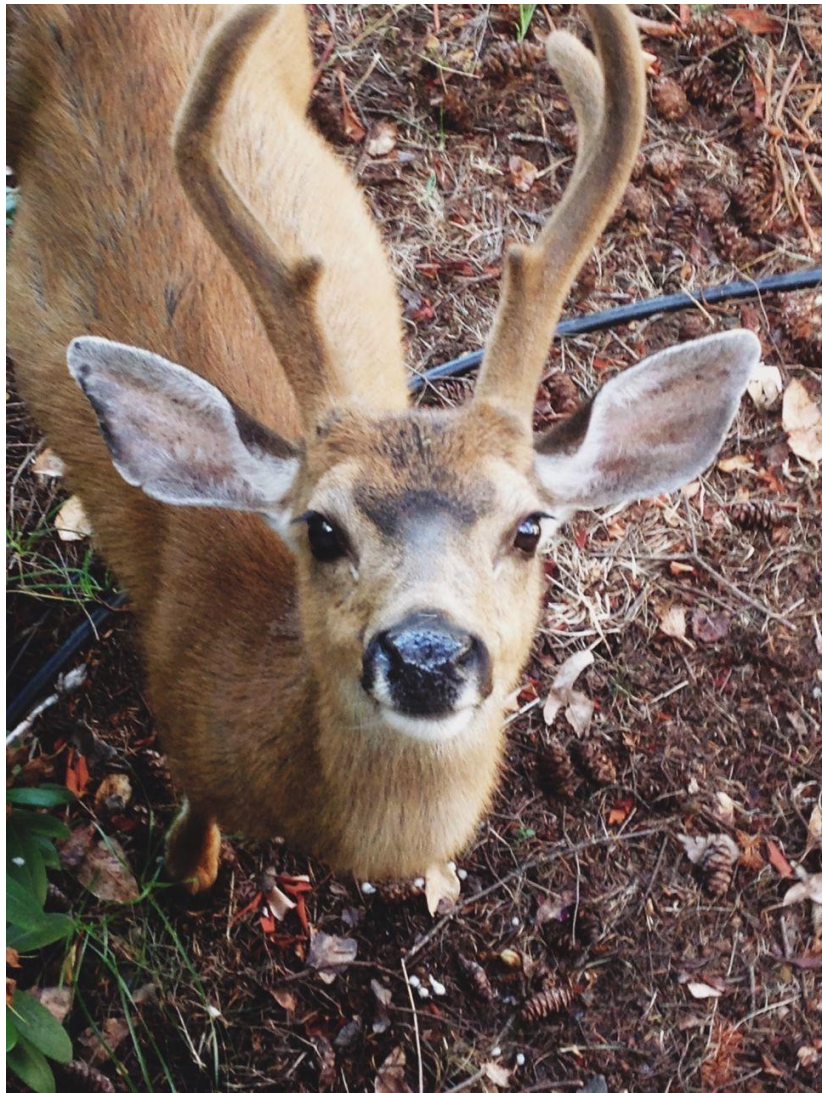



# Idiom: cancellation of several requests

```
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;
};
```

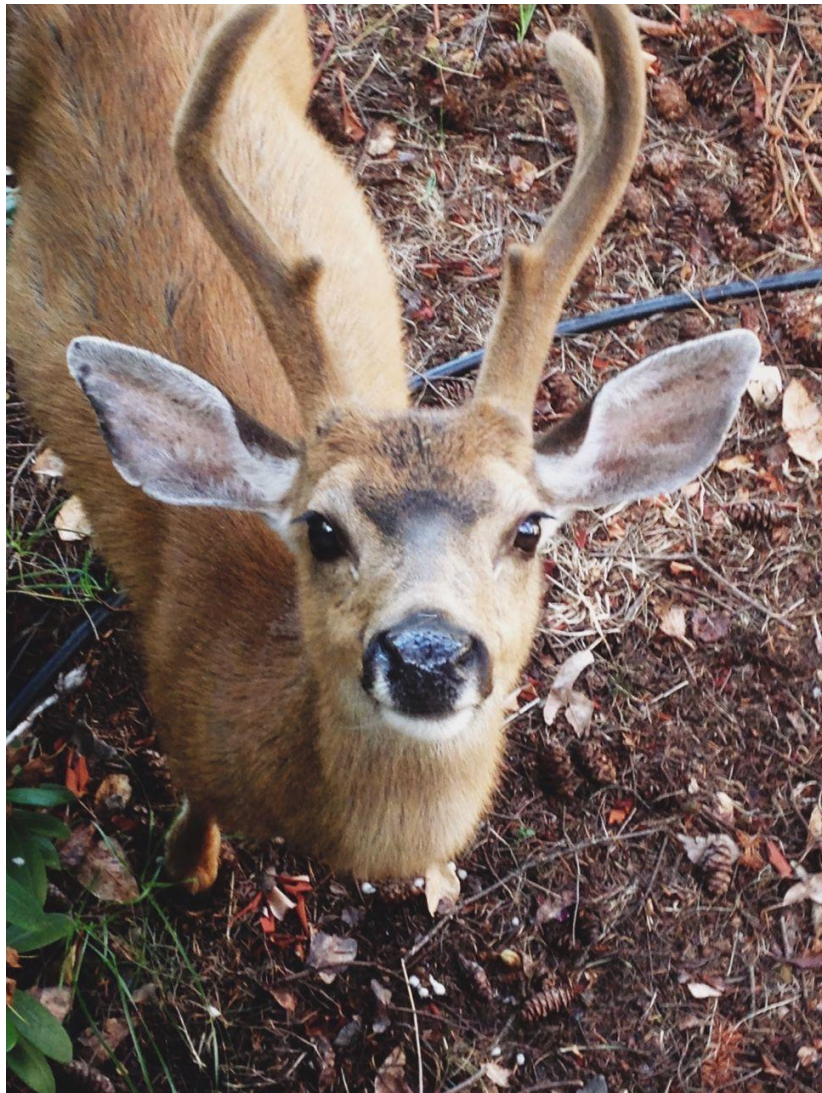
# Idiom: continuation chaining

```
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

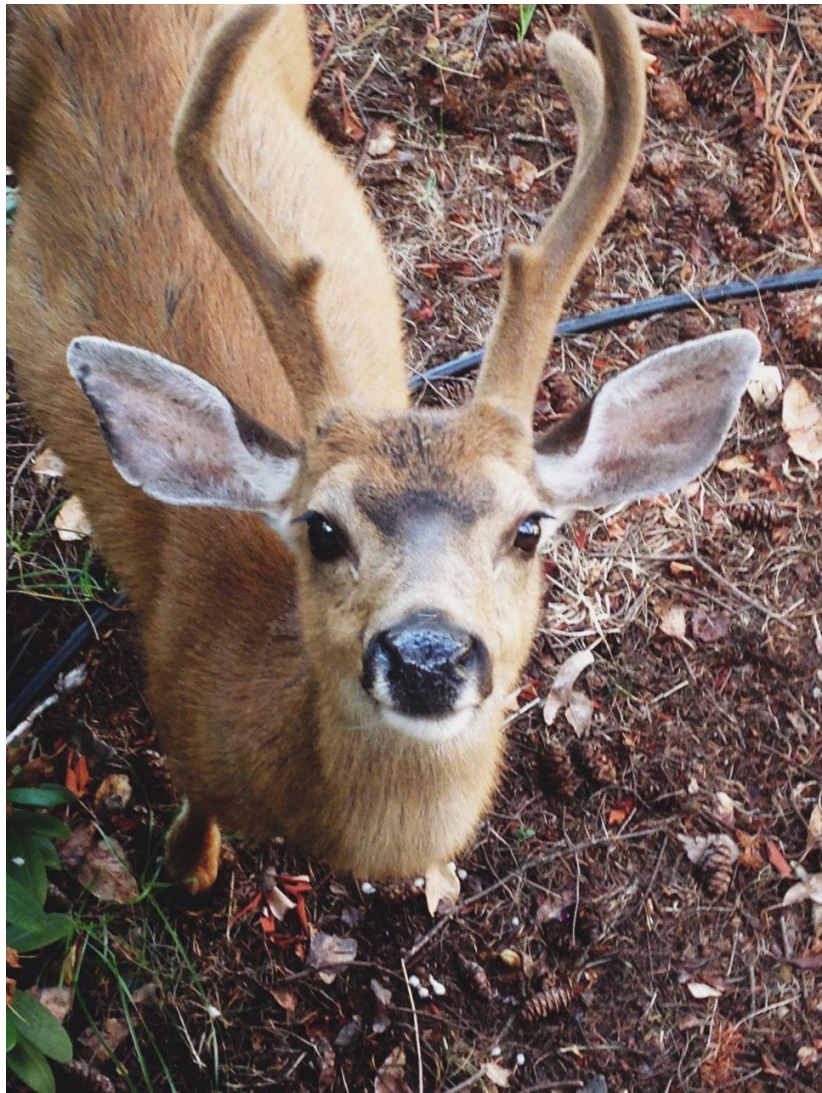
```
auto avatar = http::client::http_client{"https://reqres.in"}
  .request(http::methods::GET, "/api/users/1")
  .then([](const http::http_response &response) {
    if (response.status_code() != http::status_codes::OK)
      throw std::runtime_error("Failed to get user");
    return response.extract_json();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



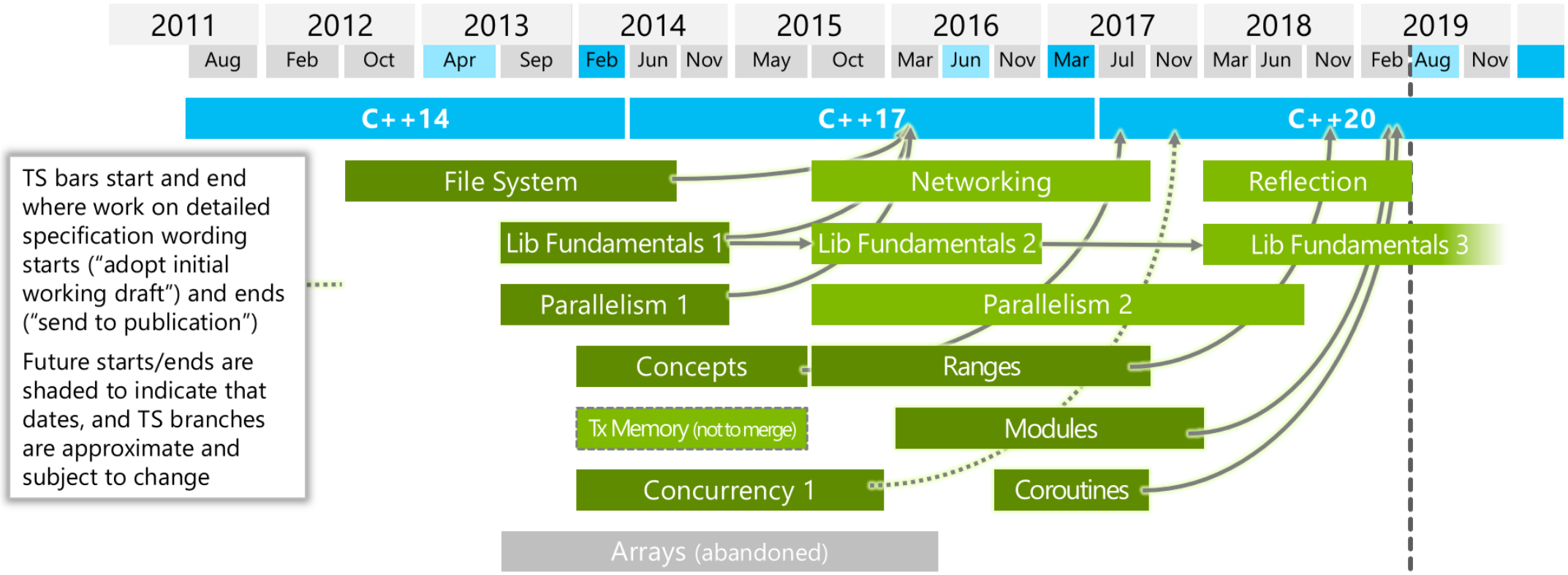
# Idiom: continuation chaining

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

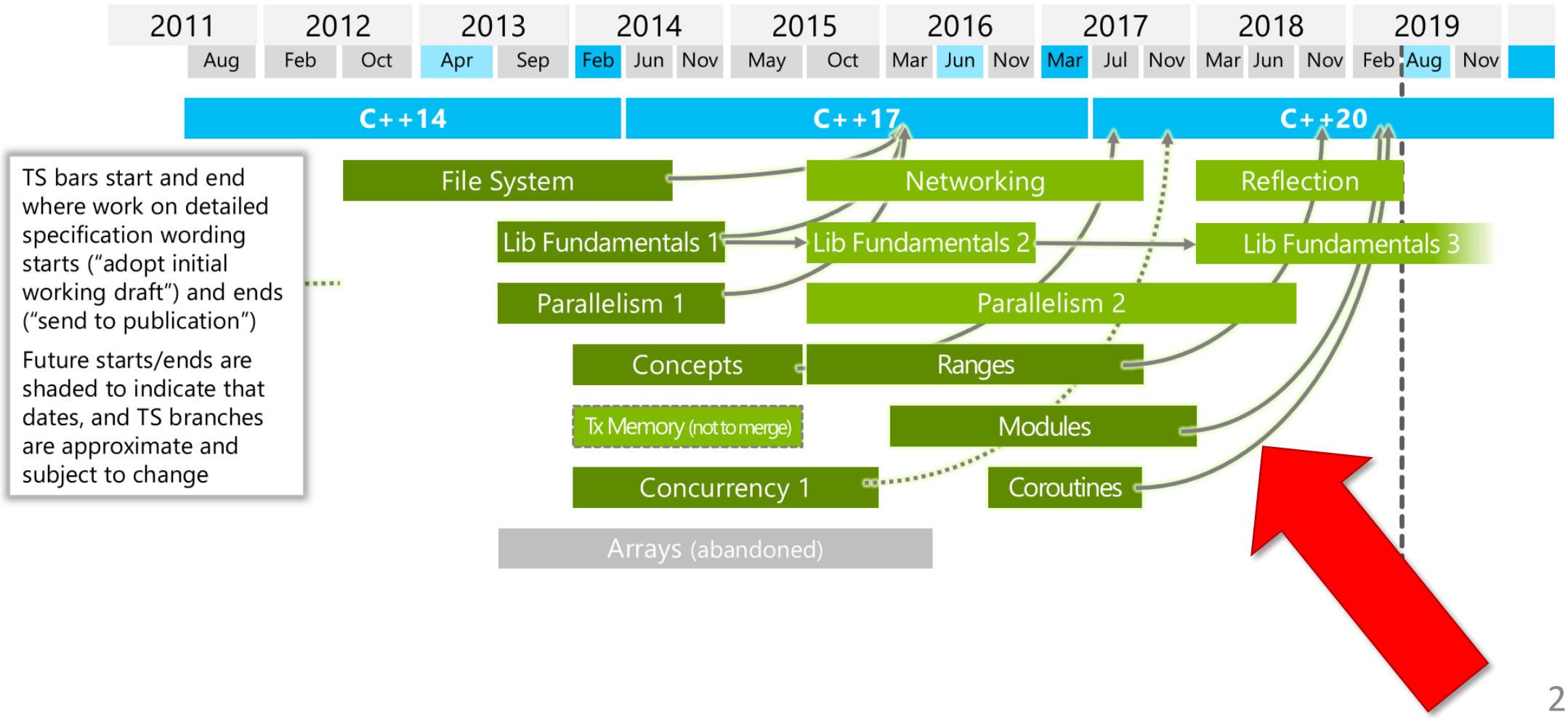
a forest puppy



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



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



# 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';
```

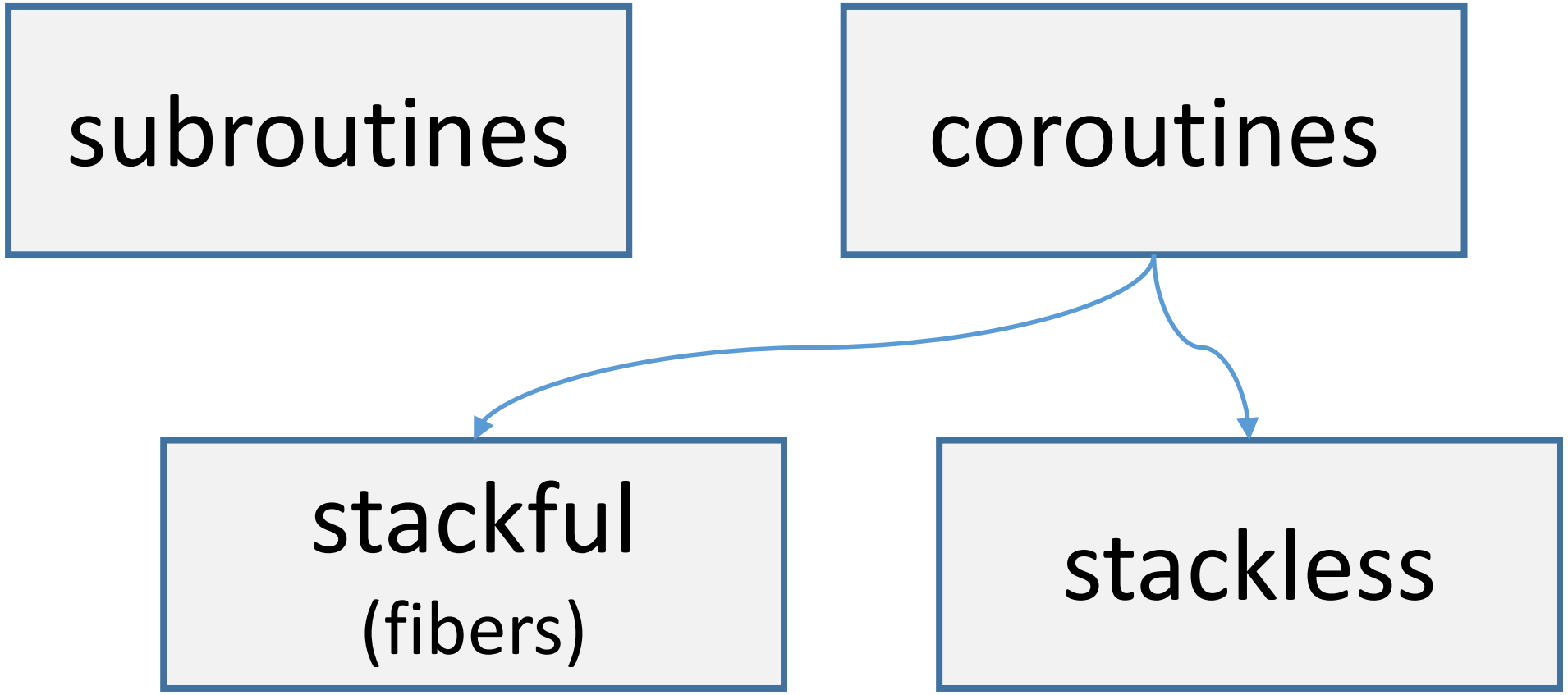
# Syntactic sugar?

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

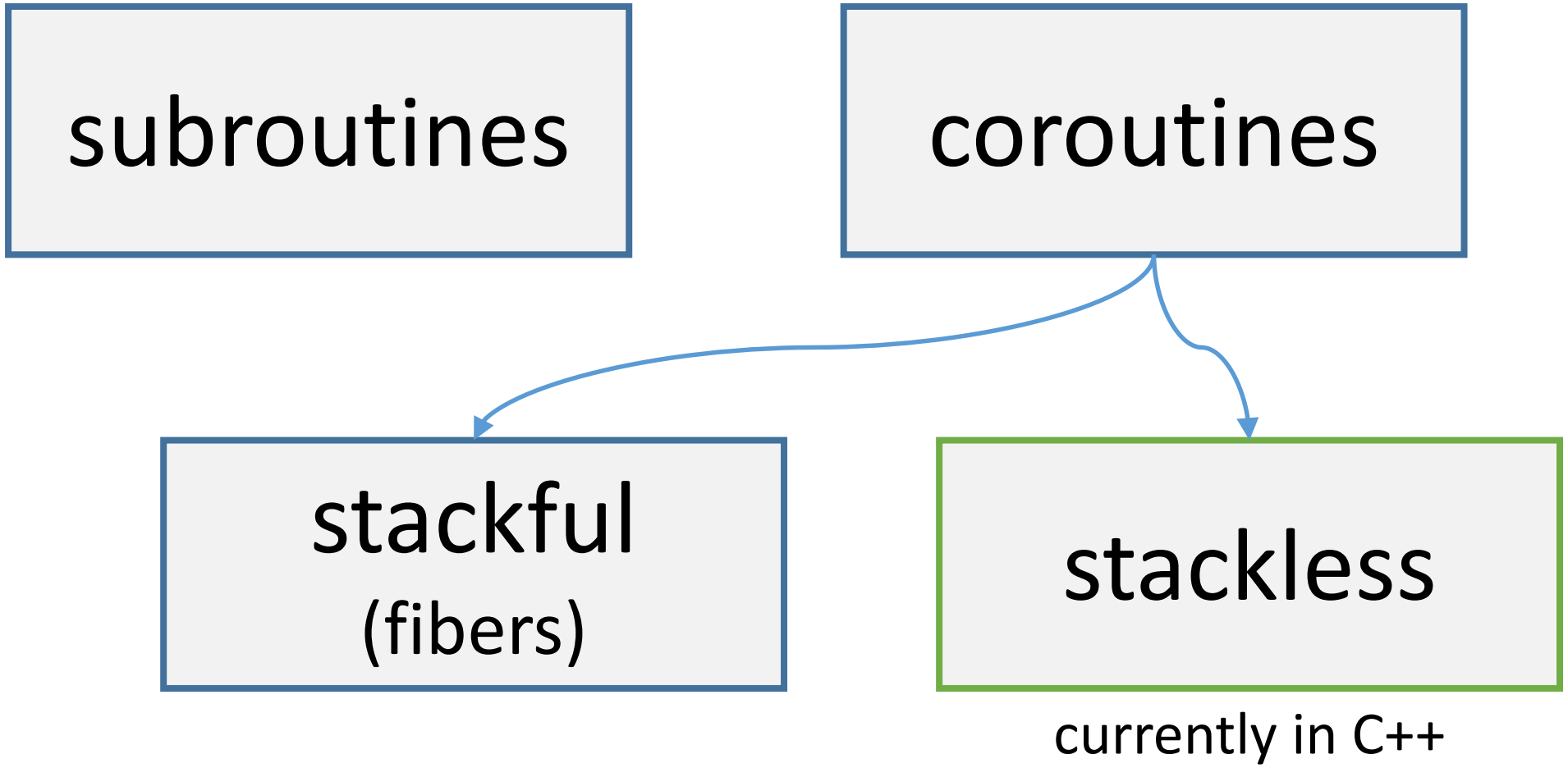
# Syntactic sugar?

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();

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

auto awaitable = operator co_await(boilWaterAsync());
if (!awaitable.await_ready()) {
  awaitable.await_suspend(coroutineHandle);
  //suspend & resume
}
awaitable.await_resume();
```

from within the coroutine context

# 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();

    sync(); //suspend & resume
    nc(); //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.unhan
final_suspend:
  co_await p.final_susp
}
```

Call stack:

boilWaterAndMakeTeaAsync$_ResumeCoro$2()

boilWaterAndMakeTeaAsync$_InitCoro$1()

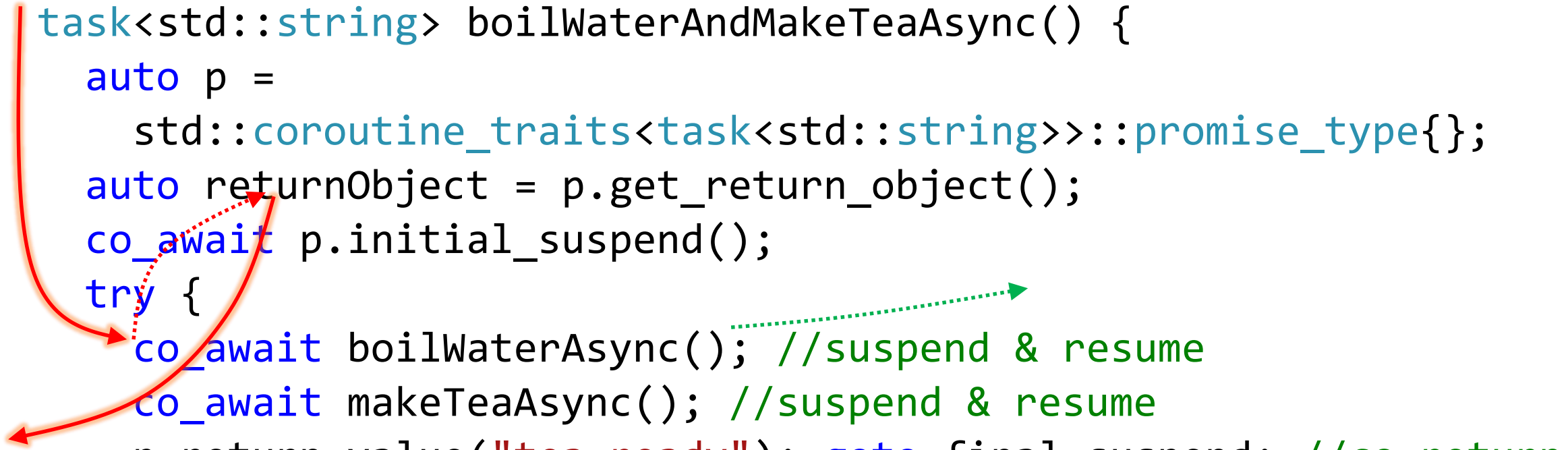
boilWaterAndMakeTeaAsync()

main(int argc, char * * argv)

# What happens "behind the curtains"

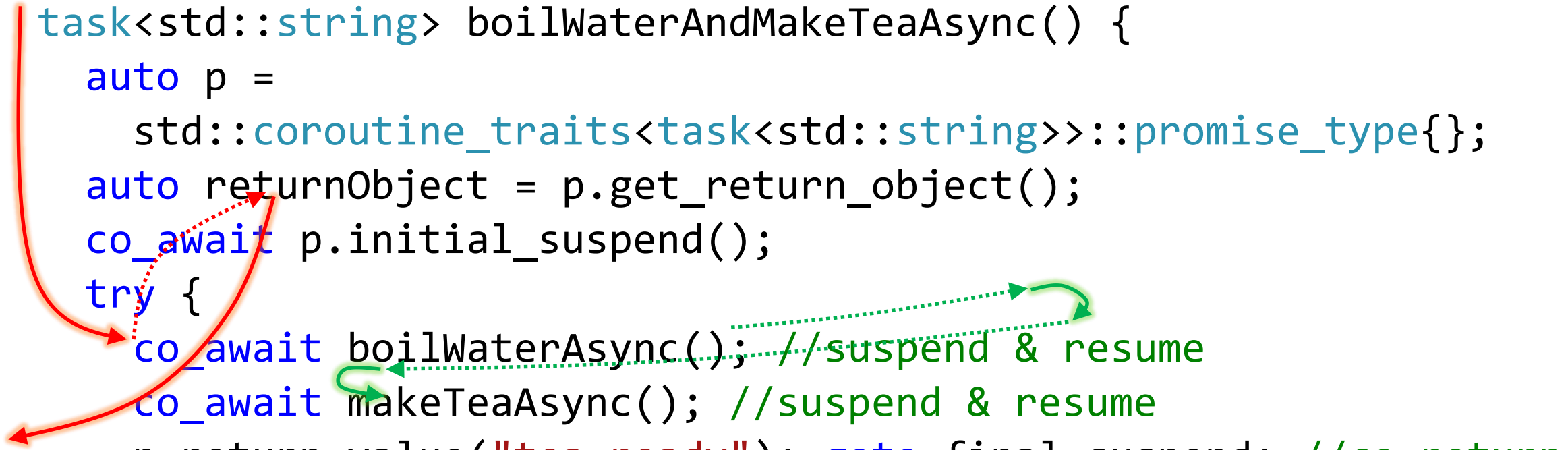
```
task<std::string> boilWaterAndMakeTeaAsync() {
  auto p =
    std::coroutine_traits<task<std::string>>::promise_type{};
  auto returnObject = p.get_return_object();
  co_await p.initial_suspend();
  try {
    co_await boilWaterAsync(); //suspend & resume
    co_await makeTeaAsync(); //suspend & resume
    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> boilW
  auto p =
    std::coroutine_trai
  auto returnObject = p
  co_await p.initial_su
  try {
    co_await boilWaterA
    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()

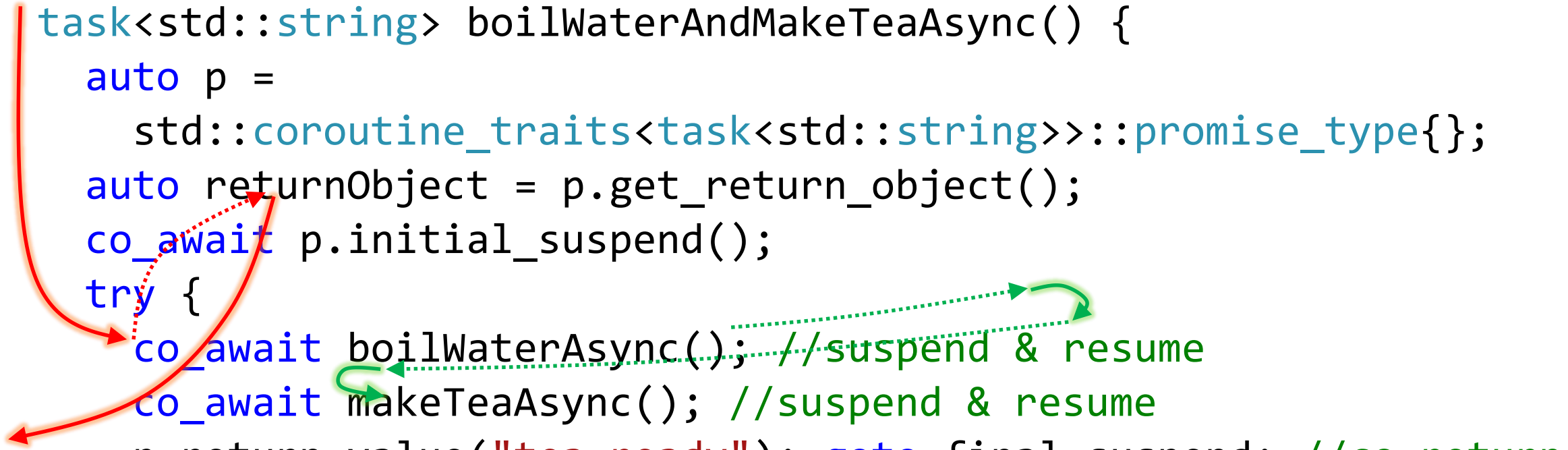
std::coroutine_handle<void>::operator()()

<task continuation>

ntdll!TppWorkpExecuteCallback()

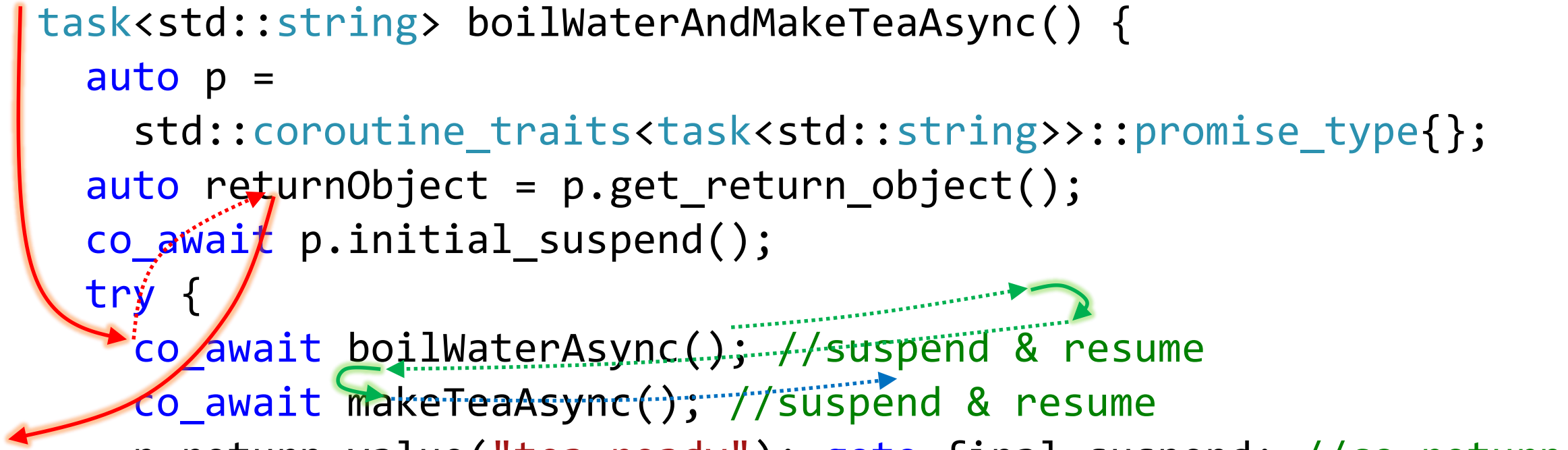
ntdll!TppWorkerThread()

# What happens "behind the curtains"



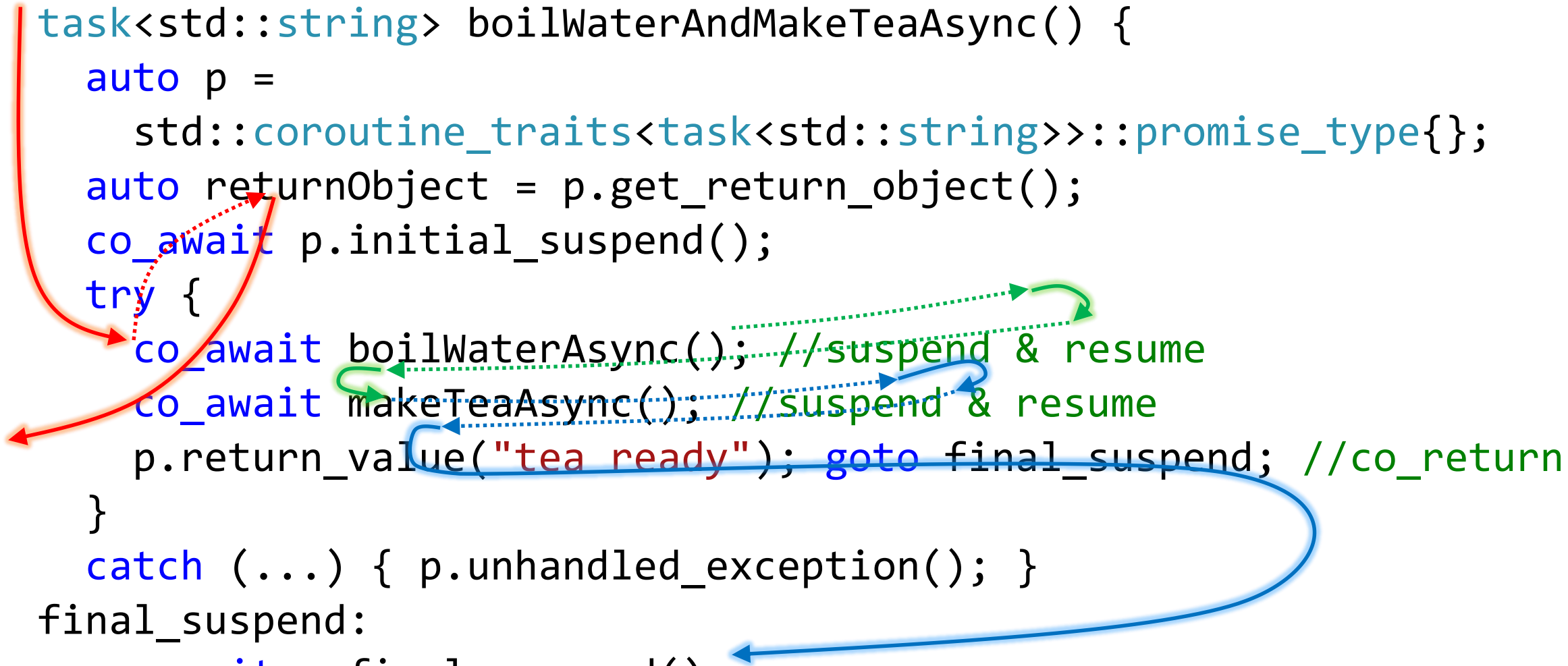
```
task<std::string> boilWaterAndMakeTeaAsync() {
  auto p =
    std::coroutine_traits<task<std::string>>::promise_type{};
  auto returnObject = p.get_return_object();
  co_await p.initial_suspend();
  try {
    co_await boilWaterAsync(); //suspend & resume
    co_await makeTeaAsync(); //suspend & resume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

# What happens "behind the curtains"

```
task<std::string> boilW
  auto p =
    std::coroutine_trai
  auto returnObject = p
  co_await p.initial_su
  try {
    co_await boilWaterA
    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()()

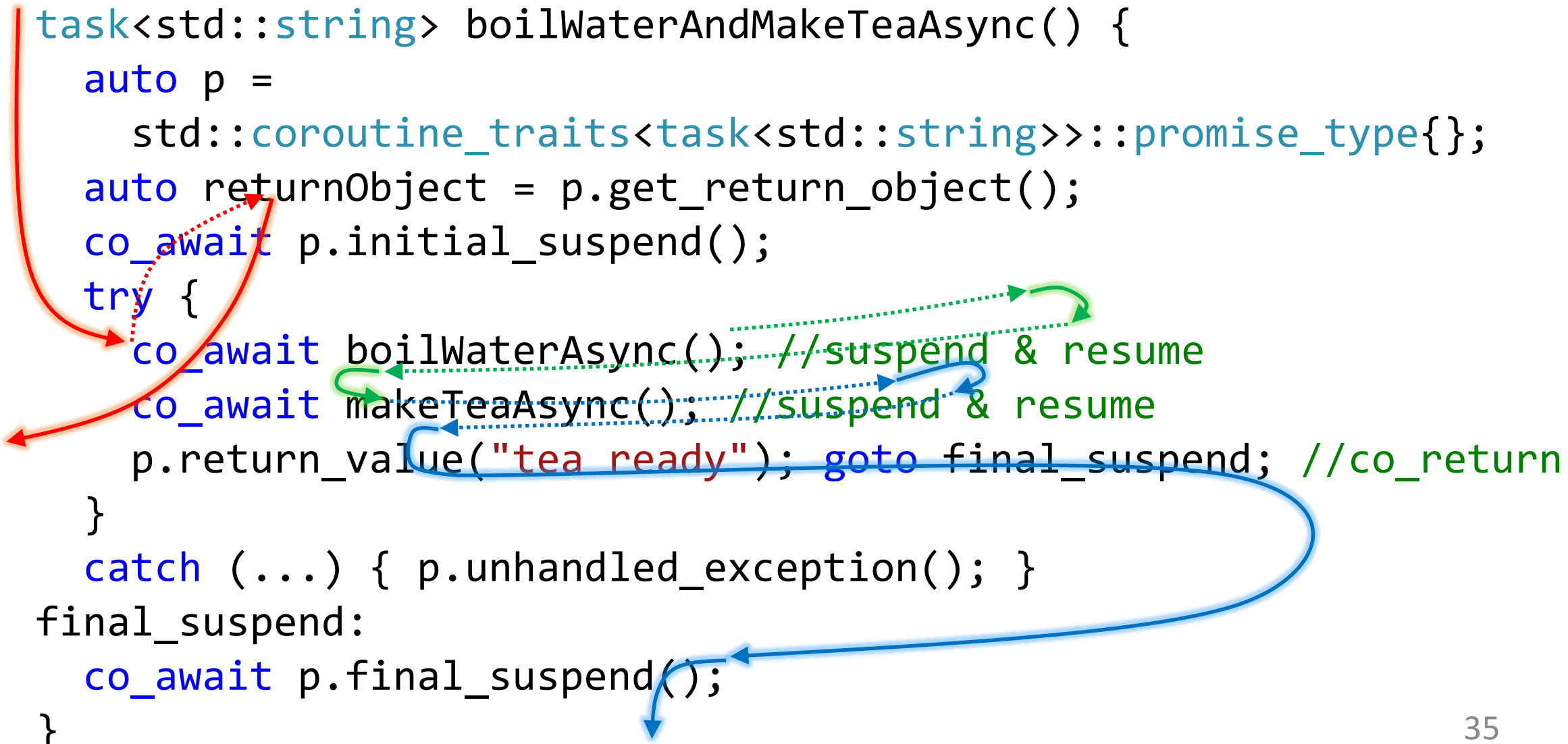
<task continuation>

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"

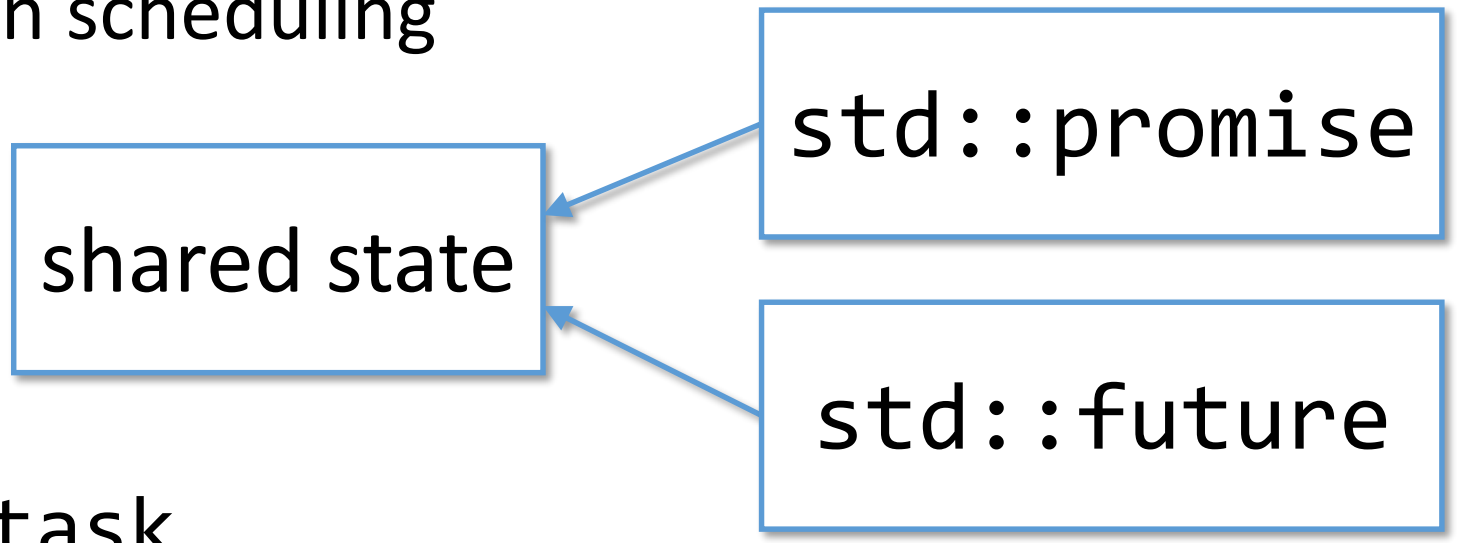
```
co_yield expression;
```

```
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

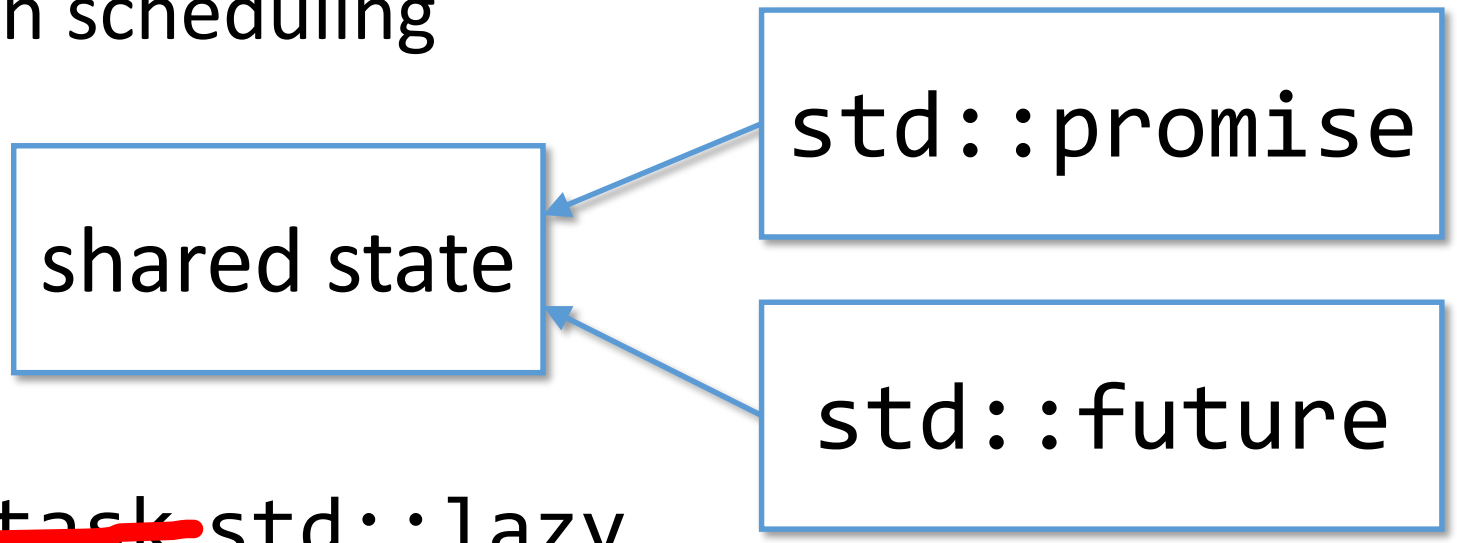



proposal P1056 in
the C++ standard: `std::task`

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


Slides: https://git.io/Jew2Y