Continuable

asynchronous programming with allocation aware futures

/Naios/continuable

Denis Blank <denis.blank@outlook.com>

Meeting C++ 2018

Introduction

About me

Denis Blank

- Master’s student @Technical University of Munich
- GSoC participant in 2017 @STEllAR-GROUP/hpx
- Author of the **continuable** and **function2** libraries
- Interested in: compiler engineering, asynchronous programming and metaprogramming

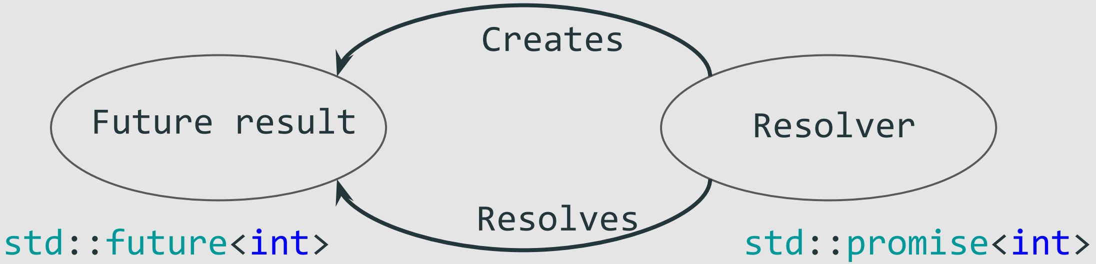
Introduction
Table of contents

The **continuable** library talk:

/Naios/continuable

The future pattern

The future pattern
promises and futures



The future pattern

Synchronous wait

```
std::promise<int> promise;
std::future<int> future = promise.get_future();

promise.set_value(42);
int result = future.get();
```

In C++17 we can only pool or wait for the result synchronously

The future pattern

Asynchronous continuation chaining

Asynchronous return types

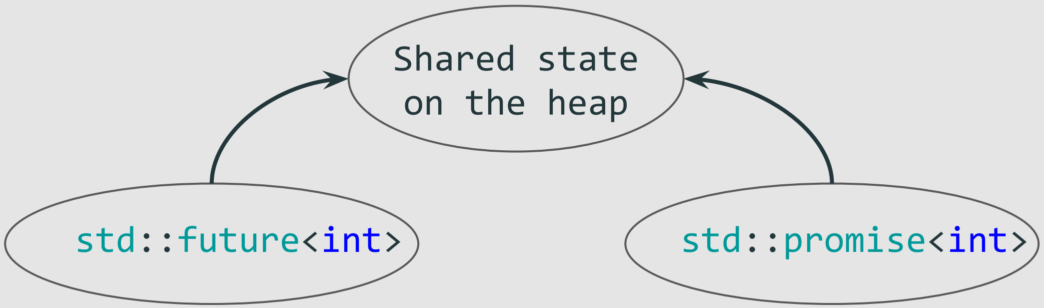
```
future<std::string> other = future
  .then([](future<int> future) {
    return std::to_string(future.get());
  });
```

Resolve the next future

The Concurrency TS proposed a then method for adding a continuation handler, now reworked in the "A Unified Futures" and executors proposal.

The future pattern

The shared state



The future pattern

Shared state implementation

```cpp
template<typename T>
class shared_state {
  std::variant<
    std::monostate, T, std::exception_ptr
  > result_;
  std::function<void(future<T>)> then_;
  std::mutex lock_;
};
```

Simplified version

The shared state contains a result storage, continuation storage and synchronization primitives.

The future pattern

Implementations with a shared state

- std::future
- boost::future
- folly::Future
- hpx::future
- stlab::future
- ...

Future disadvantages
Shared state overhead

- Attaching a continuation (then) creates a new future and shared state every time (allocation overhead)!
- Maybe allocation for the continuation as well
- Result read/write not wait free
  - Lock acquisition or spinlock
  - Can be optimized to an atomic wait free state read/write in the single producer and consumer case (non shared future/promise).
- If futures are shared across multiple cores: Shared-nothing futures can be zero cost (Seastar).

Future disadvantages
Shared state overhead

- Attaching a continuation (then) creates a new future and shared state every time (allocation overhead)!
- Maybe allocation for the continuation as well
- Result read/write not wait free
  - Lock acquisition or spinlock
  - Can be optimized to an atomic wait free state read/write in the single producer and consumer case (non shared future/promise).
- If futures are shared across multiple cores: Shared-nothing futures can be zero cost (Seastar).

Future disadvantages

Strict eager evaluation

```
std::future<std::string> future = std::async([] {
  return "Hello Meeting C++!"s;
});
```

- Futures represent the asynchronous result of an **already running** operation!
- Impossible not to request it
- Execution is non deterministic:
  - Leads to unintended side effects!
  - No ensured execution order!
- Possible: Wrapping into a lambda to achieve laziness.

Future disadvantages

Strict eager evaluation

```
std::future<std::string> future = std::async([] {
  return "Hello Meeting C++!"s;
});
```

- Futures represent the asynchronous result of an **already running** operation!
- Impossible not to request it
- Execution is non deterministic:
  - Leads to unintended side effects!
  - No ensured execution order!
- Possible: Wrapping into a lambda to achieve laziness.

Future disadvantages

Unwrapping and R-value correctness

```
future.then([] (future<std::tuple<future<int>,
                                  future<int>>> future) {
  int a = std::get<0>(future.get()).get();
  int b = std::get<1>(future.get()).get();
  return a + b;
});
```

- future::then L-value callable although consuming
  - Should be R-value callable only (for detecting misuse)
- Always required to call future::get
  - But: Fine grained exception control possible (not needed)
- Repetition of type
  - Becomes worse in compound futures (connections)

Future disadvantages

Unwrapping and R-value correctness

```
future.then([] (future<std::tuple<future<int>,
                                   future<int>>> future) {
  int a = std::get<0>(future.get()).get();
  int b = std::get<1>(future.get()).get();
  return a + b;
});
```

- future::then L-value callable although consuming
  - Should be R-value callable only (for detecting misuse)
- Always required to call future::get
  - But: Fine grained exception control possible (not needed)
- Repetition of type
  - Becomes worse in compound futures (connections)

Future disadvantages

Exception propagation

```
make_exceptional_future<int>(std::exception{})
  .then([] (future<int> future) {
    int result = future.get();
    return result;
  })
  .then([] (future<int> future) {
    int result = future.get();
    return result;
  })
```

```
.then([] (future<int> future) {
  try {
    int result = future.get();
  } catch (std::exception const& e) {
    // Handle the exception
  }
});
```

- Propagation overhead through rethrowing on `get`
- No error codes as exception type possible

Future disadvantages
Availability

Now | C++20 | C++23 | Future

- `std::future::experimental::then` will change heavily:
  - Standardization date unknown
  - “A Unified Future” proposal maybe C++23
- Other implementations require a large framework, runtime or are difficult to build

Rethinking futures

Rethinking futures
Designing goals

- Usable in a broad case of usage scenarios (boost, Qt)
- Portable, platform independent and simple to use
- Agnostic to user provided executors and runtimes
- Should resolve the previously mentioned disadvantages:
    - Shared state overhead
    - Strict eager evaluation
    - Unwrapping and R-value correctness
    - Exception propagation
    - Availability

Rethinking futures

Designing goals

- Usable in a broad case of usage scenarios (boost, Qt)
- Portable, platform independent and simple to use
- Agnostic to user provided executors and runtimes
- Should resolve the previously mentioned disadvantages:
  - Shared state overhead
  - Strict eager evaluation
  - Unwrapping and R-value correctness
  - Exception propagation
  - Availability

Rethinking futures

Why we don’t use callbacks

```
signal_set.async_wait([](auto error, int slot) {
  signal_set.async_wait([](auto error, int slot) {
    signal_set.async_wait([](auto error, int slot) {
      signal_set.async_wait([](auto error, int slot) {
        // handle the result here
      });
    });
  });
});
```

Callback hell

- Difficult to express complicated chains
- But: Simple and performant to express an asynchronous continuation.
- But: Work nicely with existing libraries

Rethinking futures

How we could use callbacks

- **Idea:** Transform the callbacks into something easier to use without the callback hell
  - Long history in JavaScript: q, bluebird
  - Much more complicated in C++ because of static typing, requires heavy metaprogramming.
- Mix this with syntactic sugar and C++ candies like operator overloading.

And finished is the **continuable**

Not trivial...

Rethinking futures
How we could use callbacks

- **Idea:** Transform the callbacks into something easier to use without the callback hell
  - Long history in JavaScript: q, bluebird
  - Much more complicated in C++ because of static typing, requires heavy metaprogramming.
- Mix this with syntactic sugar and C++ candies like operator overloading.

Not trivial...

And finished is the **continuable**

Creating continuables

Arbitrary asynchronous
return types

```
auto continuable = make_continuable<int>([](auto&& promise) {
  // Resolve the promise immediately or store
  // it for later resolution.
  promise.set_value(42);
});
```

The promise might
be moved or stored

Resolve the promise,
set_value alias for operator()

A continuable_base is creatable through make_continuable, which requires its types trough template arguments and accepts a callable type

Creating continuables

Arbitrary asynchronous return types

```
auto continuable = make_continuable<int>([](auto&& promise) {
  // Resolve the promise immediately or store
  // it for later resolution.
  promise.set_value(42);
});
```

The promise might be moved or stored

Resolve the promise, set_value alias for operator()

A continuable_base is creatable through make_continuable, which requires its types trough template arguments and accepts a callable type

Chaining continuables

Continuation chaining

```
make_ready_continuable(42)
  .then([] (int value) {
    // return something
  });
```

This ready continuable resolves the given result instantly

Optional return value:

- Plain object
- Tuple of objects
- The next continuable to resolve

A continuable_base is chainable through its then method, which accepts a continuation handler.
We work on values directly rather than continuables.

Chaining continuables

Continue from callbacks

Just a dummy function which returns a continuable_base of int, std::string

```
http_request("example.com")
  .then([] (int status, std::string body) {
    return mysql_query("SELECT * FROM `users` LIMIT 1");
  })
  .then(do_delete_caches())
  .then(do_shutdown());
```

Return the next continuable_base to resolve

Ignore previous results

then may also return plain objects, a tuple of objects or the next continuable_base to resolve.

Chaining continuables

Continue from callbacks

Just a dummy function which returns a continuable_base of int, std::string

```
http_request("example.com")
  .then([] (int status, std::string body) {
    return mysql_query("SELECT * FROM `users` LIMIT 1");
  })
  .then(do_delete_caches())
  .then(do_shutdown());
```

Return the next continuable_base to resolve

Ignore previous results

then may also return plain objects, a tuple of objects or the next continuable_base to resolve.

Chaining continuables

Continuation chaining sugar

```
make_ready_continuable('a', 2, 3)
  .then([] (char a) {
    return std::make_tuple('d', 5);
  })
  .then([] (char c, int d) {
    // ...
  });
```

Use the asynchronous arguments partially

Return multiple objects that are passed to the next continuation directly

The continuation passed to then may also accept the result partially, and may pass multiple objects wrapped inside a std::tuple to the next handler.

Chaining continuables

Continuation chaining sugar

```
make_ready_continuable('a', 2, 3)
  .then([] (char a) {
    return std::make_tuple('d', 5);
  })
  .then([] (char c, int d) {
    // ...
  });
```

Use the asynchronous arguments partially

Return multiple objects that are passed to the next continuation directly

The continuation passed to then may also accept the result partially, and may pass multiple objects wrapped inside a std::tuple to the next handler.

Continuable implementation

Continuable implementation

Creating ready continuables

```
make_ready_continuable(0, 1)
```

```
make_continuable<int, int>([] (auto&& promise) {
  promise.set_value(0, 1);
});
```

The implementation stores the arguments into a std::tuple first and sets the promise with the content of the tuple upon request (std::apply).

Continuable implementation

Decorating the continuation result

```
.then([] (auto result) {
  return;
})
```

⇨

```
.then([] (auto result) {
  return make_ready_continuable();
})
```

```
.then([] (auto result) {
  return std::make_tuple(0, 1);
})
```

⇨

```
.then([] (auto result) {
  return make_ready_continuable(0, 1);
})
```

Transform the continuation result such that it is always a continuable_base of the corresponding result.

Continuable implementation

Decorating the continuation result

```
.then([] (auto result) {
  return;
})
```

⇨

```
.then([] (auto result) {
  return make_ready_continuable();
})
```

```
.then([] (auto result) {
  return std::make_tuple(0, 1);
})
```

⇨

```
.then([] (auto result) {
  return make_ready_continuable(0, 1);
})
```

Transform the continuation result such that it is always a continuable_base of the corresponding result.

Continuable implementation

Invoker selection through tag dispatching

```
using result_t = std::invoke_result_t<Callback, Args...>;
//    ^ std::tuple<int, int> for example

auto invoker = invoker_of(identity<result_t>{});
```

3

```
// void
auto invoker_of(identity<void>);
```

1

```
// std::tuple<T...>
template<typename... T>
auto invoker_of(identity<std::tuple<T...>>);
```

2

```
// T
template<typename T>
auto invoker_of(identity<T>);
```

Continuable implementation

Attaching a continuation

```
auto continuation =
  [=](auto promise) {
    promise(1);
  };

auto callback =
  [] (int result) {
    return make_ready_continuable();
  };
```

⇨

```
auto new_continuation =
  [](auto next_callback) {
    auto proxy = decorate(callback,
                          next_callback)
    continuation(proxy);
  };
```

Attaching a callback to a continuation yields a new continuation with new argument types.

Continuable implementation

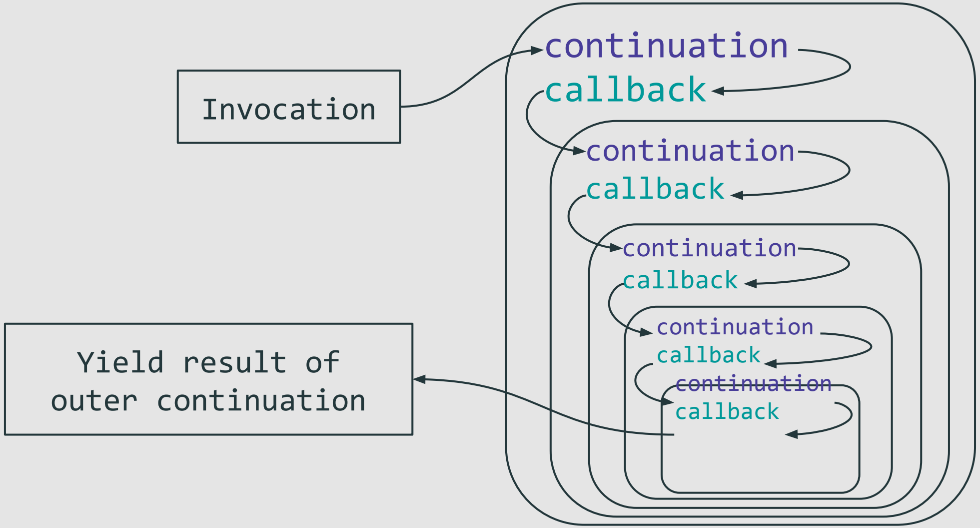
Decorating the callback

```
auto proxy = [ callback,
               next_callback ] (auto&&... args) {
  auto next_continuation = callback(std::forward<decltype(args)>(args)...);
  next_continuation(next_callback);
};
```

The proxy callback passed to the previous continuation invokes the next continuation with the next callback.

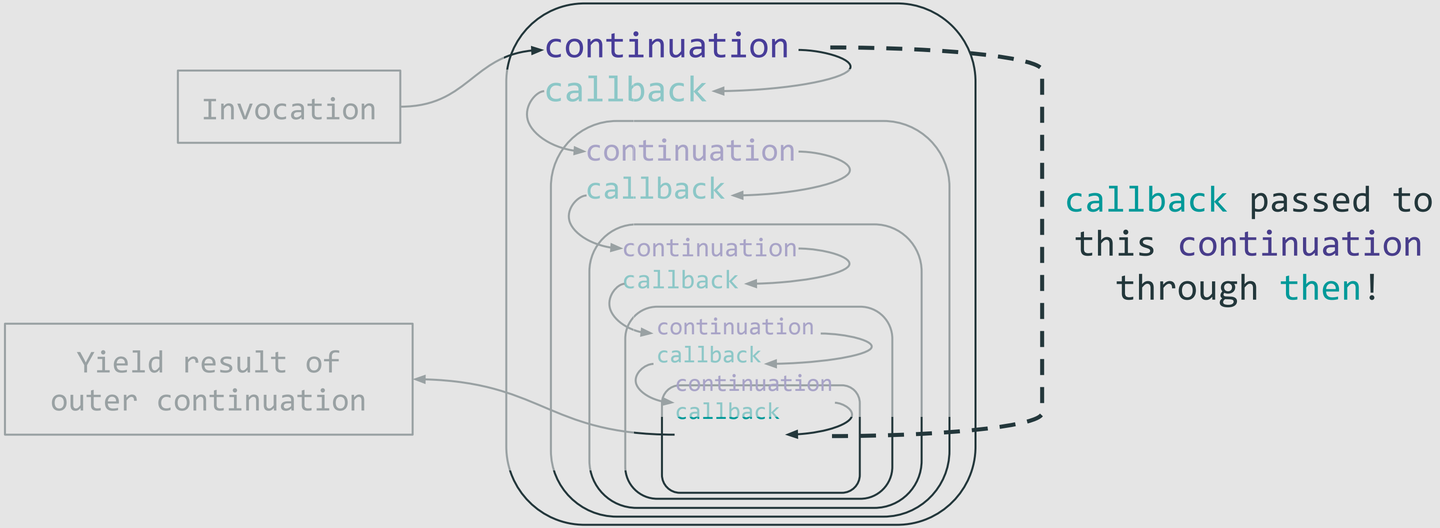
Continuable implementation

Seeing the big picture



Continuable implementation

Seeing the big picture



Continuable implementation

Seeing the big picture

Russian **Matryoshka** doll

Continuable implementation

Exception handling

```
read_file("entries.csv")
  .then([] (std::string content) {
    // ...
  })
  .fail([] (std::exception_ptr exception) {
    // handle the exception
  })
```

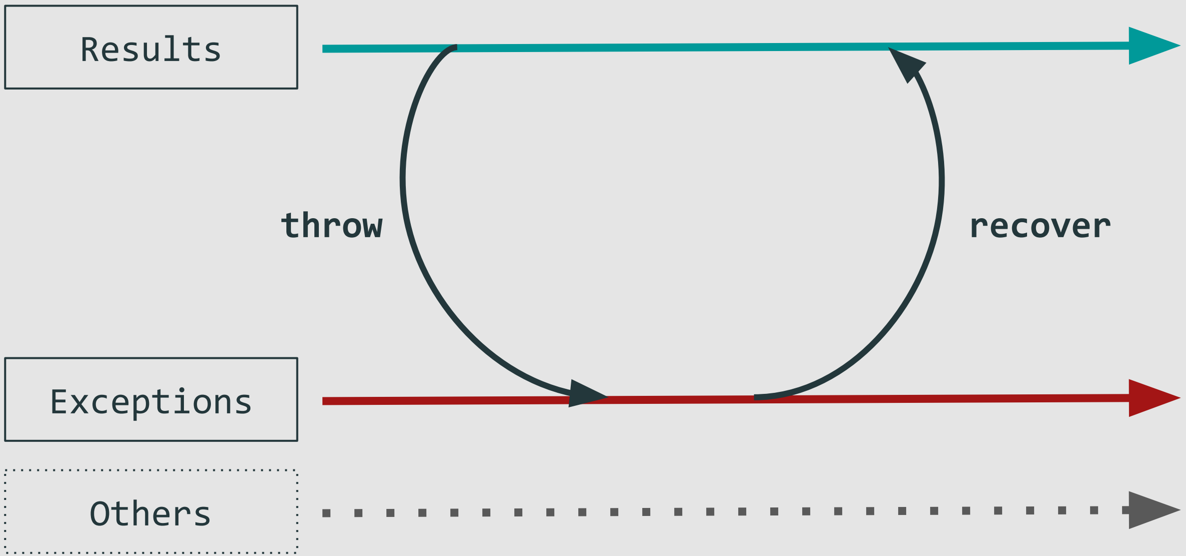
```
promise.set_exception(...)
```

On exceptions skip the result handlers between.

When the promise is resolved with an exception an exception_ptr is passed to the next available failure handler.

Continuable implementation

Split asynchronous control flows



Continuable implementation

Split asynchronous control flows

```
template<typename... Args>
struct callback {
  auto operator() (Args&&... args);

  auto operator() (dispatch_error_tag, std::exception_ptr);
  // dispatch_error_tag is exception_arg_t in the
  // "Unified Futures" standard proposal.
};
```

Or any other error type

Continuable implementation

Exception propagation

```
template<typename... Args>
struct proxy {
  Callback failure_callback_;
  NextCallback next_callback_
  void operator() (Args&&... args) {
    // The next callback has the same signature
    next_callback_(std::forward<Args>(args)...);
  }
  void operator() (dispatch_error_tag, std::exception_ptr exception) {
    failure_callback_(exception);
  }
};
```

On a valid result forward it to the next available result handler

Continuable implementation

Result handler conversion

```
template<typename... Args>
struct proxy {
  Callback callback_;
  NextCallback next_callback_
  void operator() (Args&&... args) {
    auto continuation = callback_(std::forward<Args>(args)...);
    continuation(next_callback);
  }
  void operator() (dispatch_error_tag, std::exception_ptr exception) {
    next_callback_(dispatch_error_tag{}, exception);
  }
};
```

Forward the exception to the next available handler

The continuable_base

The wrapper

```
template<typename Continuation, typename Strategy>
class continuable_base {
  Continuation continuation_;
  ownership ownership_;
  template<typename C,
           typename E = this_thread_executor>
  auto then(C&& callback,
            E&& executor = this_thread_executor{}) &&;

};
```

- void (until now)
- strategy_all_tag
- strategy_seq_tag
- strategy_any_tag

consuming = R-value

```
std::move(continuable).then(...);
```

The continuable_base is convertible when the types of Continuation are convertible to each other.

The continuable_base

The wrapper

```
template<typename Continuation, typename Strategy>
class continuable_base {
  Continuation continuation_;
  ownership ownership_;
  template<typename C,
           typename E = this_thread_executor>
  auto then(C&& callback,
            E&& executor = this_thread_executor{}) &&;

};
```

- void (until now)
- strategy_all_tag
- strategy_seq_tag
- strategy_any_tag

consuming = R-value

```
std::move(continuable).then(...);
```

The continuable_base is convertible when the types of Continuation are convertible to each other.

The continuable_base

The ownership model

```
npc->talk("Greetings traveller, how is your name?")
  .then([log, player] {
    log->info("Player {} asked for name.", player->name());
    return player->ask_for_name();
  })
  .then([](std::string name) {
    // ...
  });
```

Invoke the continuation here

The continuation is invoked when the continuable_base is still valid and being destroyed (race condition free continuation chaining).

The continuable_base
Memory allocation

Until now: no memory allocation involved!

`then` always returns an object of an unknown type

- Increases the amount of types the compiler has to generate ⇒ slower compilation
- Better for compiler optimization
- Increases the executable size
- ⇒ We require a concrete type for APIs where we don’t want to expose our implementation

The continuable_base

Concrete types

```
continuable<int, std::string> http_request(std::string url) {
  return [=](promise<int, std::string> promise) {
    // Resolve the promise later
    promise.set_value(200, "<html> ... </html>");
  };
}
```

Preserve unknown types across the continuation chaining, convert it to concrete types in APIs on request

The continuable_base

Type erasure

Erased callable for promise<Args...>

```
using callback_t = function<void(Args...),
                            void(dispatch_error_tag,
                                 std::exception_ptr)>;
```

Erased callable for continuable<Args...>

```
using continuation_t = function<void(callback_t)>;
```

For the callable type erasure my **function2** library is used that provides move only and multi signature capable type erasures + small functor optimization.

The continuable_base

Type erasure

Erased callable for promise<Args...>

```
using callback_t = function<void(Args...),
                            void(dispatch_error_tag,
                                 std::exception_ptr)>;
```

Erased callable for continuable<Args...>

```
using continuation_t = function<void(callback_t)>;
```

For the callable type erasure my **function2** library is used that provides move only and multi signature capable type erasures + small functor optimization.

The continuable_base

Type erasure aliases

```
template<typename... Args>
using promise = promise_base<callback_t<Args...>>;

template<typename... Args>
using continuable = continuable_base<
  function<void(promise<Args...>)>,
  void
>;
```

```
template<typename... Args>
using callback_t = function<void(Args...),
                            void(dispatch_error_tag,
                                 std::exception_ptr)>;
```

continuable_base type erasure works implicitly and with any type erasure wrapper out of the box.

The continuable_base

Apply type erasure when needed

```cpp
// auto do_sth();
continuable<> cont = do_sth()
  .then([] {
    return do_sth();
  })
  .then([] {
    return do_sth();
  });
```

Max 2 allocs on need

```cpp
// future<void> do_sth();
future<void> cont = do_sth()
  .then([] (future<void>) {
    return do_sth();
  })
  .then([] (future<void>) {
    return do_sth();
  });
```

2*2 fixed + 2*1 maybe continuations allocs

futures requires a minimum of two fixed allocations per then whereas continuable requires a maximum of two allocations per type erasure.

The continuable_base

Apply type erasure when needed

```cpp
// auto do_sth();
continuable<> cont = do_sth()
  .then([] {
    return do_sth();
  })
  .then([] {
    return do_sth();
  });
```

Max 2 allocs on need

```cpp
// future<void> do_sth();
future<void> cont = do_sth()
  .then([] (future<void>) {
    return do_sth();
  })
  .then([] (future<void>) {
    return do_sth();
  });
```

2*2 fixed + 2*1 maybe continuations allocs

futures requires a minimum of two fixed allocations per then whereas continuable requires a maximum of two allocations per type erasure.

Executor support

Executor support

Usage cases

```
mysql_query("SELECT `id`, `name` FROM `users` WHERE `id` = 123")
  .then([](ResultSet result) {
    // On which thread this continuation runs?
  });
```

```
promise.set_value(result);
```

- On which thread the continuation runs:
  - Resolving thread? (default)
  - Thread which created the continuation?
- When does the continuation run:
  - Immediately on resolving? (default)
  - Later?
- Can we cancel the continuation chain?

That should be up to you!

Executor support

Using an executor

```
struct my_executor_proxy {
  template<typename T>
  void operator()(T&& work) {
    std::forward<T>(work)();
  }
};

mysql_query("SELECT `id`, `name` FROM `users` WHERE `id` = 123")
  .then([](ResultSet result) {
    // Pass this continuation to my_executor
  }, my_executor_proxy{});
```

- Invoke the work
- Drop the work
- Move the work to another thread or executor

second argument of then

Executor support

No executor propagation

```
continuable<> next = do_sth().then([] {
  // Do sth.
}, my_executor_proxy{});

std::move(next).then([] {
  // No ensured propagation!
});
```

Propagation would lead to type erasure although it isn’t requested here!

The executor isn’t propagated to the next handler and has to be passed again to avoid unnecessary type erasure (we could make it a type parameter).

Executor support

Context of execution

Callback

```
do_sth().then([] {
  // Do something short
});
```

Continuation

```
continuable<> do_sth() {
  return [] (auto&& promise) {
    // Do something long
    promise.set_value();
  };
}
```

⇒ We can neglect executor propagation when moving heavy tasks to a continuation, except in case of data races!

Check design goals

- No shared state overhead ✓
- No strict eager evaluation ✓
- No unwrapping and R-value correctness ✓
- Exception propagation ✓
- Availability ✓
  - C++14
  - Header-only (depends on **function2**)
  - GCC / Clang / MSVC

Check design goals

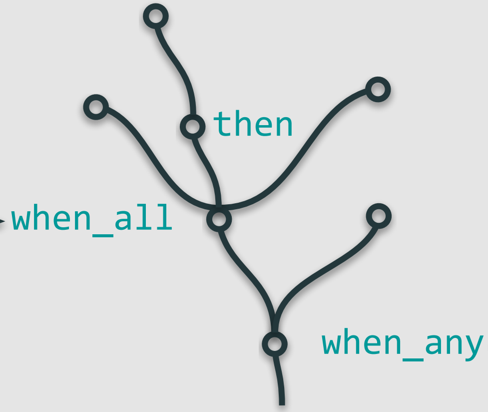
- No Shared state overhead ✔
- No strict eager evaluation ✔
- No Unwrapping and R-value correctness ✔
- Exception propagation ✔
- Availability ✔
  - C++14
  - Header-only (depends on **function2**)
  - GCC / Clang / MSVC

Connections

Connections

The call graph

Ready when <u>all</u> dependent continuables are ready



Ready when <u>any</u> dependent continuable is ready

when_all, when_any usable to express relations between multiple continuables.
⇒ Guided/graph based execution requires a shared state (not available)

Connections

Lazy evaluation advantages

Invokes all dependent continuables in sequential order

when_seq

Thoughts
(not implemented)

- when_pooled - pooling
- when_every - request all
- when_first_succeeded / when_first_failed - exception strategies

Using lazy (on request) evaluation over an eager one makes it possible to choose the evaluation strategy.
⇒ Moves this responsibility from the executor to the evaluator!

Connections

Simplification over std::experimental::when_all

```cpp
// continuable<int> do_sth();
when_all(do_sth(),
         do_sth())
  .then([] (int a, int b) {
    return a == b;
  });
```

```cpp
// std::future<int> do_sth();
std::experimental::when_all(do_sth(),
                            do_sth())
  .then([] (std::tuple<std::future<int>,
                       std::future<int>> res) {
    return std::get<0>(res).get()
        == std::get<1>(res).get();
  });
```

std::when_all introduces code overhead because of unnecessary fine grained exception handling.

Connections

Simplification over std::experimental::when_all

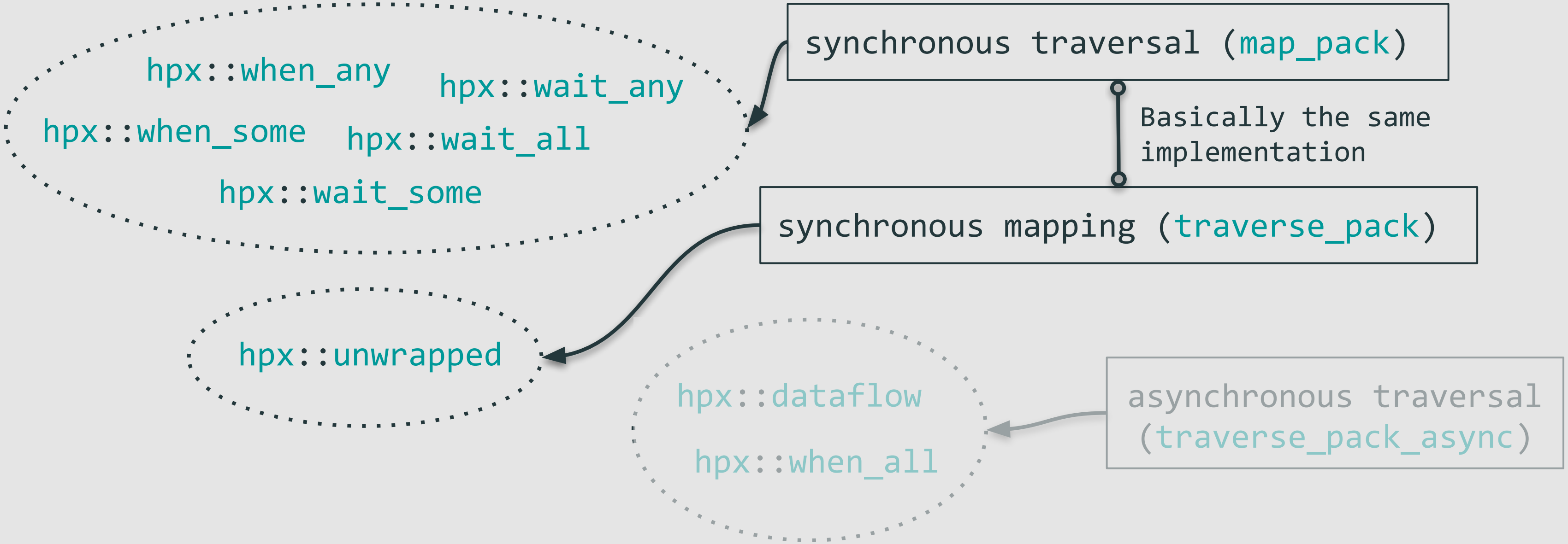
```
// continuable<int> do_sth();
when_all(do_sth(),
         do_sth())
  .then([] (int a, int b) {
    return a == b;
  });
```

```
// std::future<int> do_sth();
std::experimental::when_all(do_sth(),
                            do_sth())
  .then([] (std::tuple<std::future<int>,
                       std::future<int>> res) {
    return std::get<0>(res).get()
        == std::get<1>(res).get();
  });
```

std::when_all introduces code overhead because of unnecessary fine grained exception handling.

Connections implementation

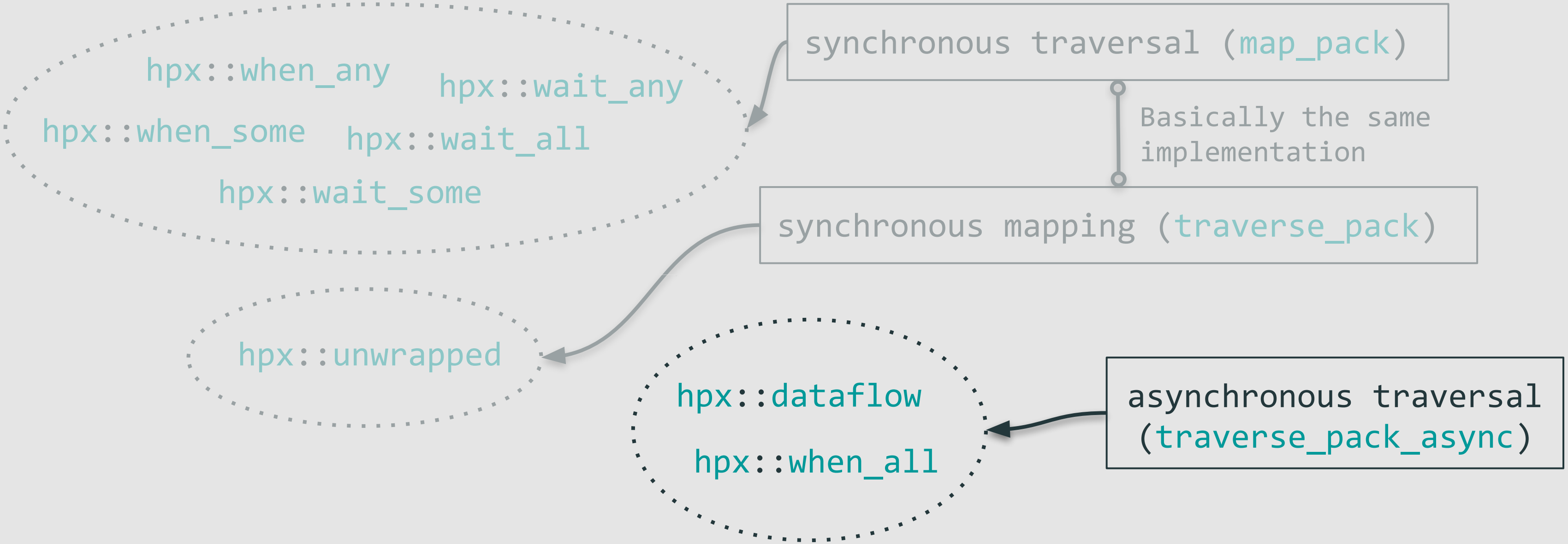
Based on GSoC @STEllAR-GROUP/hpx



The map_pack, traverse_pack and traverse_pack_async API helps to apply an arbitrary connection between continuables contained in a variadic pack.

Connections implementation

Based on GSoC @STEllAR-GROUP/hpx



The map_pack, traverse_pack and traverse_pack_async API helps to apply an arbitrary connection between continuables contained in a variadic pack.

Connections implementation

Indexer example (map_pack)

```
// continuable<int> do_sth();
when_any(do_sth(), do_sth(),
         do_sth());
  .then([] (int a) {
    // ?: We don’t know which
    // continuable became ready
  });
```

```
index_continuables(do_sth(),
                   do_sth(),
                   do_sth());


// Shall return:
std::tuple<
  continuable<size_t /*= 0*/, int>,
  continuable<size_t /*= 1*/, int>,
  continuable<size_t /*= 2*/, int>
>
```

Because when_any returns the first ready result of a common denominator, map_pack could be used to apply an index to the continuables.

Connections implementation

Indexer example (map_pack)

```
// continuable<int> do_sth();
when_any(do_sth(), do_sth(),
         do_sth());
  .then([] (int a) {
    // ?: We don't know which
    // continuable became ready
  });
```

```
index_continuables(do_sth(),
                   do_sth(),
                   do_sth());


// Shall return:
std::tuple<
  continuable<size_t /*= 0*/, int>,
  continuable<size_t /*= 1*/, int>,
  continuable<size_t /*= 2*/, int>
>
```

Because when_any returns the first ready result of a common denominator, map_pack could be used to apply an index to the continuables.

Connections implementation

Indexer example (map_pack)

```
map_pack(indexer{}, do_sth(), do_sth(), do_sth());
```

```
struct indexer {
  size_t index = 0;
  template <typename T,
            std::enable_if_t<is_continuable<std::decay_t<T>>::value>* = nullptr>
  auto operator()(T&& continuable) {
    auto current = ++index;
    return std::forward<T>(continuable).then([=] (auto&&... args) {
      return std::make_tuple(current,
                             std::forward<decltype(args)>(args)...);
    });
  }
};
```

map_pack transforms an arbitrary argument pack through a callable mapper.

Connections implementation

Indexer example (map_pack)

```
map_pack(indexer{}, do_sth(), do_sth(), do_sth());
```

```
struct indexer {
  size_t index = 0;
  template <typename T,
            std::enable_if_t<is_continuable<std::decay_t<T>>::value>* = nullptr>
  auto operator()(T&& continuable) {
    auto current = ++index;
    return std::forward<T>(continuable).then([=] (auto&&... args) {
      return std::make_tuple(current,
                             std::forward<decltype(args)>(args)...);
    });
  }
};
```

map_pack transforms an arbitrary argument pack through a callable mapper.

Connections implementation

Arbitrary and nested arguments

```
continuable<int> aggregate(std::tuple<int,
                                      continuable<int>,
                                      std::vector<continuable<int>>> all) {

  return when_all(std::move(all))
    .then([] (std::tuple<int, int, std::vector<int>> result) {
      int aggregated = 0;
      traverse_pack([&] (int current) {
        aggregated += current;
      }, std::move(result));
      return aggregated;
    });
}
```

map_pack and friends can work with plain values and nested packs too and so can when_all.

Connections implementation

Arbitrary and nested arguments

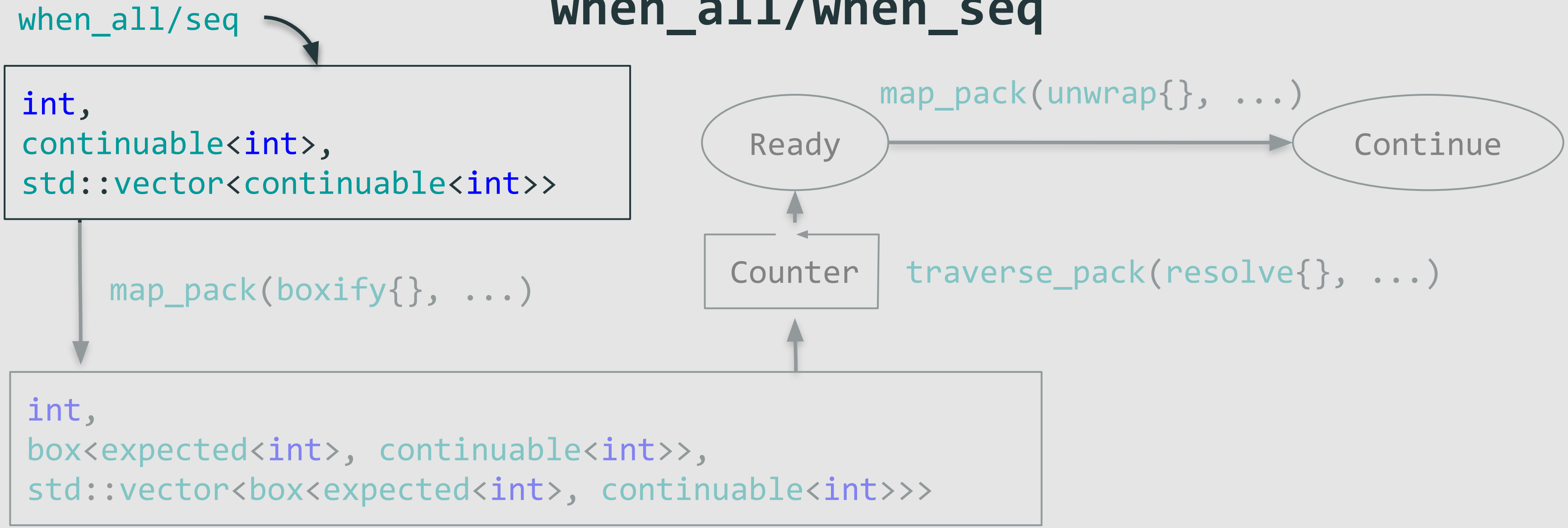
```
continuable<int> aggregate(std::tuple<int,
                                      continuable<int>,
                                      std::vector<continuable<int>>> all) {

  return when_all(std::move(all))
    .then([] (std::tuple<int, int, std::vector<int>> result) {
      int aggregated = 0;
      traverse_pack([&] (int current) {
        aggregated += current;
      }, std::move(result));
      return aggregated;
    });
}
```

map_pack and friends can work with plain values and nested packs too and so can when_all.

Connections implementation

when_all/when_seq



Connections require a shared state by design,
concurrent writes to the same box never happen.

Connections implementation

when_all/when_seq

when_all/seq

```
int,
continuable<int>,
std::vector<continuable<int>>
```

```
map_pack(boxify{}, ...)
```

```
int,
box<expected<int>, continuable<int>>,
std::vector<box<expected<int>, continuable<int>>>
```

Counter

```
traverse_pack(resolve{}, ...)
```

Ready

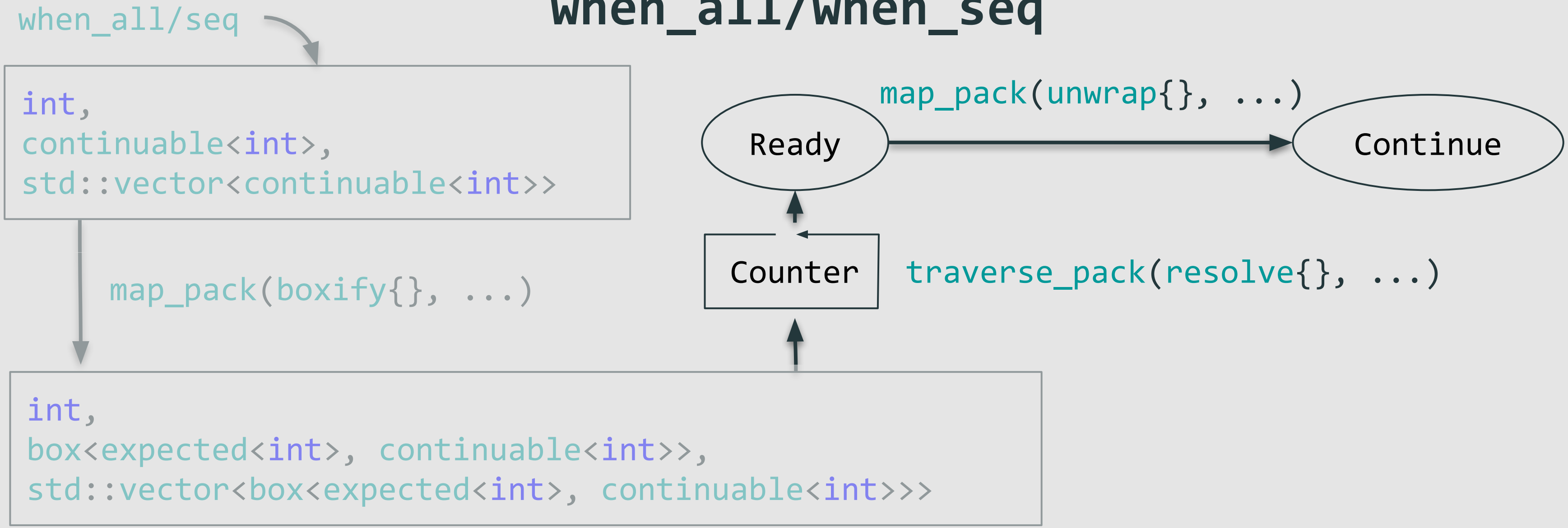
```
map_pack(unwrap{}, ...)
```

Continue

Connections require a shared state by design,
concurrent writes to the same box never happen.

Connections implementation

when_all/when_seq



Connections require a shared state by design,
concurrent writes to the same box never happen.

Operator overloading

Express connections

when_all: operator&&

```
(http_request("example.com/a") && http_request("example.com/b"))
  .then([] (http_response a, http_response b) {
    // ...
    return wait_until(20s)
        || wait_key_pressed(KEY_SPACE)
        || wait_key_pressed(KEY_ENTER);
  });
```

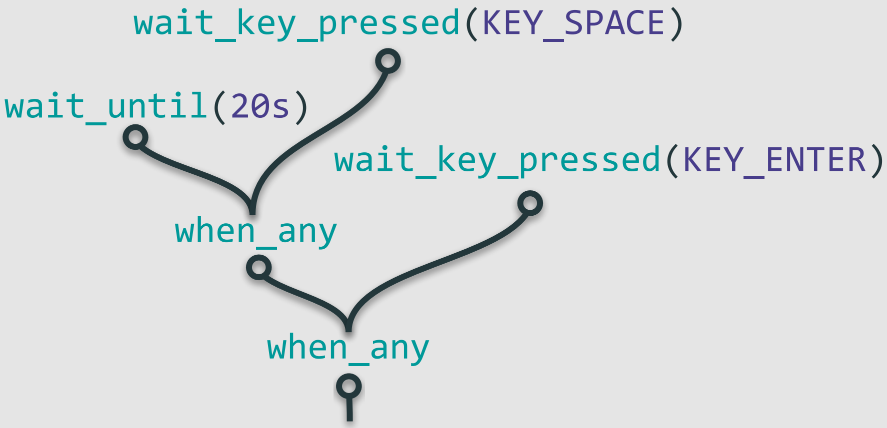
when_any: operator||

Operator overloading allows expressive connections between continuables.

Operator overloading

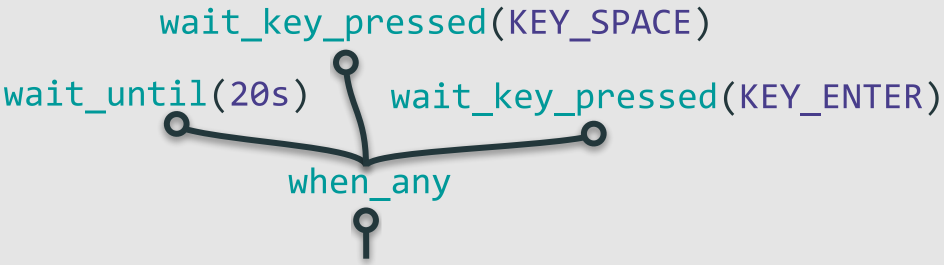
Difficulties

```
return wait_until(20s)
     || wait_key_pressed(KEY_SPACE)
     || wait_key_pressed(KEY_ENTER);
```



A naive operator overloading approach where we instantly connect 2 continuables would lead to unintended evaluations and thus requires linearization.

Operator overloading
Correct operator evaluation required



Operator overloading

Implementation

```
wait_until(20s)
|| wait_key_pressed(KEY_SPACE)
```

⇨

```
continuable_base<std::tuple<..., ...>,
                 strategy_any_tag>
```

```
... || wait_key_pressed(KEY_SPACE)
```

⇨

```
continuable_base<std::tuple<..., ..., ...>,
                 strategy_any_tag>
```

```
.then(...)
```

$\xrightarrow{\text{Materialization}}$

```
continuable_base<..., void>
```

Set the `continuable_base` into an intermediate state (strategy), materialize the connection on use or when the strategy changes (expression template).

Continuable & Coroutines TS

Continuable & Coroutines TS

Interoperability

```
continuable<int> interoperability_check() {
  try {
    auto response = co_await http_request("example.com/c");
  } catch (std::exception const& e) {
    co_return 0;
  }
  auto other = cti::make_ready_continuable(0, 1);
  auto [ first, second ] = co_await std::move(other);
  co_return first + second;
}
```

continuable_base implements operator co_await() and specializes coroutine_traits and thus is compatible to the Coroutines TS.

Continuable & Coroutines TS

Interoperability

```
continuable<int> interoperability_check() {
  try {
    auto response = co_await http_request("example.com/c");
  } catch (std::exception const& e) {
    co_return 0;
  }
  auto other = cti::make_ready_continuable(0, 1);
  auto [ first, second ] = co_await std::move(other);
  co_return first + second;
}
```

continuable_base implements operator co_await() and specializes coroutine_traits and thus is compatible to the Coroutines TS.

Continuable & Coroutines TS

Do Coroutines deprecate Continuables?

Probably not!

There are many things a plain coroutine doesn’t provide

- A coroutine isn’t necessarily allocation free
  - Recursive coroutines frames
  - Depends on compiler optimization
- Connections
- Executors (difficult to do with plain coroutines)
- Takes time until Coroutines are widely supported
  - Libraries that work with plain callbacks (legacy codebases)
- But: Coroutines have much better Call Stacks!

Continuable & Coroutines TS

Do Coroutines deprecate Continuables?

Probably not!

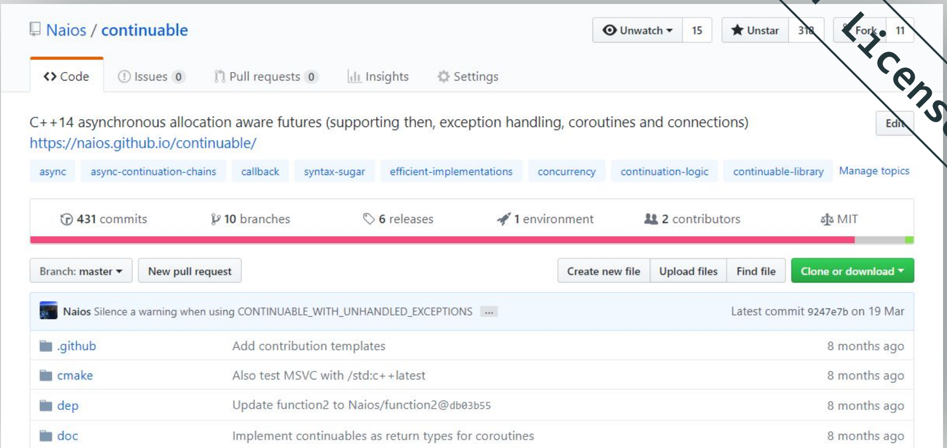
There are many things a plain coroutine doesn’t provide

- A coroutine isn’t necessarily allocation free
  - Recursive coroutines frames
  - Depends on compiler optimization
- Connections
- Executors (difficult to do with plain coroutines)
- Takes time until Coroutines are widely supported
  - Libraries that work with plain callbacks (legacy codebases)
- But: Coroutines have much better Call Stacks!

Questions?

Thank you for your attention

MIT Licensed



slides

/Naios/talks

Denis Blank <denis.blank@outlook.com>

me

code

/Naios/continuable