Continuable

asynchronous programming with allocation aware futures

/Naios/continuable

Denis Blank <denis.blank@outlook.com>

Meeting C++ 2018
Introduction

About me

- 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

1. The future pattern (and its disadvantages)
2. Rethinking futures
   - Continuable implementation
   - Usage examples of continuable
3. Connections
   - Traversals for arbitrarily nested packs
   - Expressing connections with continuable
4. Coroutines
The future pattern
The future pattern promises and futures

```
std::future<int>
```

```
std::promise<int>
```

Future result

Resolver

Creates

Resolves
The future pattern

Synchronous wait

```cpp
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

---

In C++17 we can only pool or wait for the result synchronously.
The future pattern
Asynchronous continuation chaining

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

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

Shared state on the heap

std::future<int>

std::promise<int>
The future pattern
Shared state implementation

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

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 (\texttt{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

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

- 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

```cpp
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

- `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

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

```c
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
            });
        });
    });
});
```
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**...
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*
A `continuable_base` is creatable through `make_continuable`, which requires its types through template arguments and accepts a callable type.
Creating continuables

```cpp
class continuable {
public:
    // Resolve the promise immediately or store it for later resolution.
    template <typename T>
    T make_continuable(auto&& promise) {
        promise.set_value(42);
    }
};
```

A `continuable` 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 continuatable resolves the given result instantly

Optional return value:
- Plain object
- Tuple of objects
- The next continuatable 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#endif
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) {
    // ...
  });

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

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

```cpp
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

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

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

Decorating the continuation result

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

Transform the continuation result such that it is always a `continuable_base` of the corresponding result.
Continuable implementation
Invoker selection through tag dispatching

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

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

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

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

3. // void
   auto invoker_of(identity<void>);
Continuable implementation
Attaching a continuation

```cpp
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

Invocation

Yield result of outer continuation
Continuable implementation

Seeing the big picture

Invocation

continuation

callback

continuation

callback

continuation

callback

continuation

callback

continuation

callback

callback passed to this continuation through then!

Yield result of outer continuation
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
    })

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

promise.set_exception(...)

On exceptions skip the result handlers between.
Continuable implementation
Split asynchronous control flows

- Results
- Exceptions
- Others
Continuable implementation
Split asynchronous control flows

```cpp
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);
    }
};
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 \texttt{continuable\_base} \\
The wrapper

\begin{verbatim}

\texttt{template<typename Continuation, typename Strategy>}
\texttt{class continuable\_base {}
    Continuation continuation_{};
    ownership ownership_{};
\texttt{template<typename C,}
\texttt{    typename E = this\_thread\_executor>}
\texttt{auto then(C&& callback,}
\texttt{    E&& executor = this\_thread\_executor{}}) &&;
};
\texttt{consuming = R-value}
\texttt{std::move(continuable).then(...);}

The \texttt{continuable\_base} is convertible when the types
of Continuation are convertible to each other.
\end{verbatim}
The continuable_base

The wrapper

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 \(\Rightarrow\) slower compilation
- Better for compiler optimization
- Increases the executable size
- \(\Rightarrow\) We require a concrete type for APIs where we don’t want to expose our implementation
Concrete types

The continuable_base

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

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

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

for promise<Args...>

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

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**

```cpp
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

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(); });

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

**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

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

```cpp
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

```cpp
continuable<> next = do_sth().then([] { // Do sth.
    // No ensured propagation!
    });

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

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).
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

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

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!

Thoughts
(not implemented)

- when_pooled - pooling
- when_every - request all
- when_first_succeeded / when_first_failed - exception strategies
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;
  });

// 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

```cpp
// continuables<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.
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)

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

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

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

```cpp
// Shall return:
std::tuple<
    continuables<int> do_sth();
continuable<size_t /*= 0*/, int>,
continuable<size_t /*= 1*/, int>,
continuable<size_t /*= 2*/, int>
>
```
Connections implementation  
Indexer example (map_pack)

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

```cpp
// continuables<int> do_something();
when_any(do_something(), do_something(),
         do_something());
.then([] (int a)
   // ?: We don’t know which
   // continuables became ready
   )

index_continuables(do_something(),
                   do_something(),
                   do_something());

// Shall return:
std::tuple<
    continuables<
        size_t /*= 0*/ , int>,
    continuables<
        size_t /*= 1*/ , int>,
    continuables<
        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)**

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

```cpp
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)

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

```cpp
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

```cpp
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

```cpp
continuable<int> aggregate(std::tuple<int, continuabile<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 require a shared state by design, concurrent writes to the same box never happen.
Connections require a shared state by design, concurrent writes to the same box never happen.
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.
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
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

```cpp
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

Denis Blank <denis.blank@outlook.com>