Callbacks and Composition

Document #: P1678
Date: 2019-05-18
Project: Programming Language C++
SG1 Concurrency and Parallelism, LEWG Library Evolution
Reply-to: Kirk Shoop
<kirkshoop@fb.com>

Contents

1 Introduction

A goal of this paper is to select one callback pattern that can be used by default for functions and callbacks being added to C++ libraries. There will still be Invocables added that should not conform to this pattern. For example, The whole purpose of algorithms is to apply Invocable projections and Invocable predicates to data.

A non-goal of this paper is to explore or select the callback pattern for callback sequences. This is a goal of a later paper and this paper has been written with callback sequences in mind. callback sequences will be the basis of AsyncRange proposals, work well for streaming data over a network and also will apply to audio and ui events.

> NOTE: callbacks are just as likely to be synchronous as asynchronous. While this paper will use an async function to explore the different styles for functions and callbacks - the motivations and proposals in this paper apply to both synchronous and asynchronous functions taking callbacks.

1.1 examples of callbacks

callbacks are common in the standard library. `std::visit` and `std::for_each` are algorithms that take a callback. The `std::thread` constructor takes a callback. `std::promise` **is** a callback, and the proposed `std::experimental::future::then()` takes a callback. callbacks are fundamental to the Networking TS and the Executors proposal. C++20 coroutines generate callbacks and a state-machine for calling them so that users can write code without explicit callbacks.

The callback pattern is so common because a function call is a great way to transfer data (eg. the function passed to `std::visit`), change execution context (eg. the function passed to `std::thread`) and modify the behaviour of an algorithm (eg. the predicate passed to `std::sort`).

1.2 `async_accept` example

Examples of callbacks used for async functions can be found in the Networking TS [N4771]. For the purpose of comparison, one async function and its completion signature will be used to explore the different designs presented in this paper. The signature for `async_accept()`, as defined in [N4771] is:

```
template<
    typename SocketService,
    typename AcceptHandler>
void-or-deduced async_accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint,
    AcceptHandler handler);
```

This signature indicates that there is a single function where the last argument is used for the callback that will be provided the result and the preceding arguments contain the inputs needed to generate the result.

The completion signature for `async_accept()` is `void(error_code ec, socket_type s)`. This completion signature indicates that there is a single function where the first argument is used for the error and the second argument is used for the result that will be called for errors and results.

1.3 composing callbacks

All of the functions and callbacks mentioned have different signatures. The primary effect of all these disparate signatures is that composition of one function and callback to another, say `std::promise` and `void(error_code ec, socket_type s)`, must be written by hand. For all the functions and callbacks that already exist, this manual adaptation cannot be avoided. Proposing a solution that is composable, is the motivation for this paper.

2 Motivation

The STL ended a long period where each implementation of a list or dynamic array or string had a unique surface. The different representations of containers and element traversal shared enough terminology to cause confusion and shared enough semantics to be dangerous when composed with algorithms and other container implementations.

Functions taking callbacks, at the moment, are exactly where containers were before the STL. Just like the STL needed to define stable Container & Range concepts to be used to compose different containers and algorithms together, C++ needs to define stable Sender & Callback concepts to be used to compose different tasks and executors and algorithms together.

2.1 composition challenges

2.1.1 callbacks

Callbacks are challenging for composition because they all have a different shape. Signatures such as:

— completion/termination - `void()`
— error and values - `void(auto ec, auto... v)`
— errors - `void(auto ec)`
— values - `void(auto... v)`

All these callback patterns exist and make composition of callbacks a bespoke, repetitive, and error-prone task.

A callback that takes error and value arguments, must support invalid or empty states for each argument, because the same function will be called for error and success. When error and result are delivered to the same function all implementations of these callbacks are required to check the arguments for validity before using the arguments. These checks introduce branches, which can be particularly expensive instructions.

Another way to represent an empty state is to use `std::optional` explicitly on all the args so that the value types used as callback arguments are not required to support an invalid or empty state, but the branches remain the same and the codegen for `std::optional` is added.

> NOTE: `std::error_code` supports an 'empty' state. The empty state for a `std::error_code` is the success code.

2.1.2 functions taking callbacks

A friction point for functions that take callbacks, is that the callback is placed in different positions in the argument list of the function:

— first - `std::visit(callback, variant...)`

— last - `std::for_each(begin(r), end(r), callback)`

These different callback argument patterns exist and make composition of functions taking callbacks a bespoke, repetitive and error-prone task.

A limitation of functions, that take callbacks as the last argument, is that this affects valid signatures for and overloading of the functions. Having a fixed last argument, requires adding overloads when defaulting the values of the non-callback arguments. Having a fixed last argument, requires that the overloads be constrainable so that the callback can be reliably distinguished from the non-callback arguments. Having a fixed last argument, prevents using variadic non-callback arguments, which is why `std::visit` places the callback as the first argument.

2.2 algorithm composition

Composition is improved when callbacks have a regular shape. A regular shape for callbacks and functions taking callbacks allows composition to be generically implemented in algorithms.

If all callbacks had a regular shape then it would allow algorithms like `std::this_thread::sync_get`, `std::when_all`, `std::when_any`, `std::retry`, `std::repeat`, `std::take_until`, `std::timeout`, `std::at`, `std::sequenced`, `std::on`, `std::via`, `std:just`, `std::defer`, `std::transform`, etc.. to compose different functions taking callbacks.

Such composition might result in code that looked like this:

Table 1: using `at`, `timeout` and `when_any` to get fresh data and update a cache or fallback to the cache if the data takes too long

function

```
auto get_data() {
  auto fallback = sequenced(
    at(ex, now() + 4s),
    cached_request(ex));
  return timeout(
    when_any(
      update_cache(network_request(ex)),
      move(fallback)),
    at(ex, now() + 5s));
}
```

pipe operator

```
auto get_data() {
  auto fallback = at(ex, now() + 4s) |
    sequenced(cached_request(ex));
  return when_any(
      network_request(ex) | update_cache(),
      move(fallback)) |
    timeout(at(ex, now() + 5s));
}
```

Table 2: using `when_all` to compose async (`get_data`) and sync (`std::visit`) callbacks with `retry` and `take_until`

function	pipe operator
`auto foo(stop_token s) {`	`auto foo(stop_token s) {`
`  return take_until(when_all(`	`  return when_all(`
`      retry(get_data(), 3),`	`      get_data() | retry(3),`
`      visit(v_)),`	`      visit(v_)) |`
`    s);`	`    take_until(s);`
`}`	`}`

Table 3: using `defer` and `repeat` to keep a connection alive until cancelled

function	pipe operator
`auto keep_alive(stop_token s) {`	`auto keep_alive(stop_token s) {`
`  return take_until(`	`  return defer([ex](){`
`      repeat(sequenced(`	`      return at(ex, now() + 5s); }) |`
`        defer([ex](){`	`    sequenced(ping_request(ex)) |`
`          return at(ex, now() + 5s); }),`	`    repeat() |`
`        ping_request(ex)))`	`    take_until(s);`
`    s);`	`}`
`}`	

3 Function output

Here is a short description of the options currently in the language for functions to return values. These options boil down to three channels; return value, out-parameter arguments, and throwing exceptions.

3.1 Values

In C, there are three ways to communicate a result:

— return a value
— set value(s) into out-parameter(s)
— call a parameter, that is a function, with arguments(s)

3.2 Exceptions

C++ added a third mechanism for communicating a result - throwing exceptions. Adding exception throwing as a separate communication channel allowed code to focus on the path of success and delegate the responsibility for exception handling to the caller by default. C++ made support for exceptions implicit.

Functions do not have a mechanism to opt-in to exception support. Functions can opt out of emitting exceptions using `noexcept`, but the compiler still is responsible for ensuring that an attempt to throw an exception in a `noexcept` function will result in a call to `std::terminate`.

3.3 Multiplexing

These mechanisms can be multiplexed and de-multiplexed, with additional overhead in code size and runtime.

Examples of mux for return values and out-parameters:

— `optional<T>` allows return without a result.
— `expected<E, T>` allows an error to be returned without an exception.
— `expected<E, optional<T>>` allows an error to be returned without an exception and for nothing to be returned.
— `expected<optional<variant<tuple<Tn0...>, tuple<Tn1...>, ..>>, E>` allows the parameters that are supported by one of an overload set of callback functions to be returned as a value and an error to be returned without an exception and for nothing to be returned.

Potential syntax to simplify the code that needs to be written to demux these values can be found in the proposal for pattern matching [P1371R0].

NOTE: while `expected`, `variant` and `tuple` all have corresponding C++ language features (exception & return value have `expected`, overload set of functions have `variant`, and multiple arguments to a function have `tuple`), `optional` does not have a language representation. Pointer is not a language representation as`optional` is a super-set of Pointer, because `optional` stores the value when it is valid, while Pointer does not.

4 Representation

4.1 callback

There are infinite representations of callbacks. A few of these will be described in this paper and will be explored using `async_accept()` as defined in [N4771] and its completion signature `void(error_code ec, socket_type s)`.

4.1.1 value and error arguments style

Using separate arguments to a callback to represent error and value channels involves some unfortunate tradeoffs. The completion signature `void(error_code ec, socket_type s)` for `async_accept()` in [N4771] implies that the `socket_type` must support an invalid or empty state when ec contains an error. This style requires that all the parameters used in a completion signature support invalid or empty states, because the same function will be called for error and success. This requires all implementations of callbacks to check the arguments for validity before using the arguments. These checks introduce branches, which can be particularly expensive instructions.

4.1.2 completion token style

The Networking TS [N4771] uses the completion tokens described in [N4045] to overload the callback argument to async functions like `async_accept()`. If the callback argument is a function matching the completion signature (eg. `void(error_code ec, socket_type s)` for `async_accept()`), then the function is used as the callback (in other words, the completion token style subsumes the value and error arguments style). On the other hand, if the argument is a completion token, then the completion token implements a callback that matches the completion signature for the async function and uses the implementation of that callback to convert from value and error arguments style to some other callback representation (eg. `std::promise` type:).

Therefore, a completion token is a callback adaptor factory and a callback is an Invocable. [N4045] describes some machinery to hide the differences while implementing an async function.

[N4045] 9.1.1 contains this example for implementing an async function using machinery to hide the differences:

```
template <class Buffers, class CompletionToken>
auto async_foo(socket& s, Buffers b, CompletionToken&& token) {
  async_completion<CompletionToken,
    void(error_code, size_t)> completion(token);
  // ..
  return completion.result.get();
}
```

[N4045] 9.2 discusses the implementation of a block completion type that uses `std::future::get()` to block the caller until the result is available. The implementation of the completion token is way too long to include here, but here is the usage for `async_accept` with the `block` completion token:

```
socket_type t = async_accept(socket, endpoint, block);
```

With [N4045], the callback form would be something like:

```
async_accept(socket, endpoint, [](error_code ec, socket_type s){});
```

With [N4045], the future & get form would be something like:

```
socket_type t = async_accept(socket, endpoint, use_future).get();
```

With [N4045], the coroutine form would be something like:

```
socket_type t = co_await async_accept(socket, endpoint, use_coroutine);
```

Having `async_accept()` take so many forms does affect reading code and writing generic code. Sometimes `async_accept()` returns the result, sometimes an object, and sometimes void, readers will need to adjust to seeing the same function take on different forms. Generic code is more complicated - to wrap up an async function like `async_accept()` in a generic function, the generic function must pick a completion token to use while customizing the call to the async function but the completion token that the generic function picks might not compose well or efficiently with the completion token passed to the generic function that must ultimately communicate the result to the caller.

Underneath each of the forms generated by an async function that uses a completion token would be a function of the value and error arguments style that was implemented by the completion token. As mentioned, the value and error arguments style requires that all arguments have an invalid or empty state.

The completion token appears to add some additional constraints on the completion signature. One constraint is that, if there is an error argument, it must be the first argument. Another constraint is that the error argument must be the `error_code` type. These constraints seem to be necessary because completion tokens like `use_future` and `use_coroutine` need to be able to distinguish between the error argument and the result argument(s), and need to be able to depend on how to convert the error argument into a thrown exception. Library functions can be built to generalize these additional constraints, much as completion tokens themselves are used to overload the meaning of a callback argument to an async function.

4.1.3 `std::expected` style

Another callback pattern is to combine the value and error into one argument. The completion signature for the `async_accept()` example might change to look something like `void(expected<error_code, socket_type> e)`.

This style does not require `socket_type` to support an invalid or empty state because it does not need to be constructed when there is an error. The branches required by the value and error arguments style are still required in this style, because the same function will be called for error and success.

There is also an additional cost in the codegen for packing and unpacking `std::expected`. The cost for `std::expected` is not as bad as when the value is a `std::tuple` or a `std::variant` of `std::tuple`s, but still worse than when it is an plain argument to the function. For instance, something that transforms the result from one type to another has to check the error, unpack the result or error and repack the transformed result or original error into the outgoing expected type.

4.1.4 multiple function style

Some of the tradeoffs encountered when mixing errors and results into the same 'channel' (where function arguments and function results are both channels for communication with a function), motivated the creation of the C++ exception channel. C++ exceptions do not require the implementation of a function to check for the validity of function return values before using them and do not require that function return values support invalid or empty states (basically re-implementing `std::optional` in each type) nor require the use of types that combine error/value alternatives like `std::expected`.

Using multiple functions for error and result is equivalent to the separation of `return value` and `throw`/`catch` in the language. Using multiple functions for error and result produces very different tradeoffs than when mixing error and result together in one function. The `std::promise` type: is an example of using multiple functions for error and result that already exists.

4.1.4.1 `std::promise` type:

The `std::promise` type provides the member functions `set_value(T)`|`set_value()` and `set_exception(std::exception_pt`

— `set_value(T)`|`set_value()` is only called when there is result. Thus `T` does not need to support an invalid or empty state and implementations of `set_value` are not required to check for errors and thus no branch is added for that check.
— `set_exception(std::exception_ptr)` is only called when there is an error. Thus implementations of `set_exception` are not required to check for success and thus no branch is added for that check.

4.1.4.2 concepts:

A challenge with the `std::promise` type: is that it is a type with only one implementation, whereas callbacks are intended to be a concept or signature with many implementations. There are several examples of concepts that use multiple functions for error and result. These concepts primarily differ only in the names of the concepts and the names of the functions.

— Reactive Extensions defines the Observer concept which has been implemented in many different languages including C++. The rxcpp implementation uses the names `Observer::on_next(T)`, `Observer::on_error(std::exception_ptr)` and `Observer::on_completed()`
— [P1055R0] defines the Single concept using the names `Single::value(T)`, `Single::error(E)` and `Single::done()`

— [P1341R0] defines the Receiver concept using the names `Receiver::value(Tn...)`, `Receiver::error(E)` and `Receiver::done()`. The pushmi library has an implementation of the Receiver concept.
— [P1660] defines the Callback concept that subsumes the Invocable and Fallback concepts resulting in the names `Invocable::operator()(Tn...)`, `Fallback::error(E)` and `Fallback::done()`. [P1660] includes an example implementation.

NOTE: see [P1677] which contains a justification, for the existence of, and some uses for, the `done()` method.

The Callback concept defined in [P1660] has been gaining support in SG1 recently. A completion object for the `async_accept()` example might change to look something like:

```
struct async_accept_completion {
  void operator()(socket_type s) && noexcept;
  void error(error_code) && noexcept;
  void error(exception_ptr) && noexcept;
  void done() && noexcept;
};
```

Where:

— `operator()` is only called for success
— `error()` is only called for failure
— `done()` is only called for neither-a-result-nor-an-error (see [P1677])

With [P1660] the Callback form would be something like:

```
std::submit(
  async_accept(socket, endpoint),
  std::callback(
    [](socket_type s){},
    [](error_code ec){}));
```

With [P1660] the future & get form would be something like:

```
socket_type t = std::future_from(async_accept(socket, endpoint)).get();
```

With [P1660] the coroutine form would be something like:

```
socket_type t = co_await async_accept(socket, endpoint);
```

In all of these `async_accept` does not change form. `async_accept` always returns a type that matches a concept that other functions can adapt. The adaption is not a new mechanism that is injected into the implementation of `async_accept`. Adaption is just a function that is passed the result of `async_accept` and adapts it to some other callback representation. `std::submit`, `std::future_from`, and `operator co_await()` are all external to the `async_accept` function and each of those functions has a stable form across all usage.

4.2 function taking callback

In practice there appear to be very few representations of functions taking callbacks. These few will be described in this paper and will be explored using `async_accept()` and its signature, as defined in [N4771]:

```
template<
    typename SocketService,
    typename AcceptHandler>
```

```
void-or-deduced async_accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint,
    AcceptHandler handler);
```

4.2.1 function argument style

Using separate arguments to a function to represent callback and input channels involves some unfortunate tradeoffs. The signature for `async_accept()` defined in [N4771] as it might look if only an Invocable callback argument was supported:

```
template<
    typename SocketService,
    typename Fn>
void async_accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint,
    Fn handler);
```

The `async_accept()` signature accepts an Invocable callback as the last argument. The preceding arguments are inputs needed to produce the result.

When the Invocable callback is an argument, there is a need to be able to distinguish the input arguments from the Invocable callback argument. C++ has no way to universally inspect the arguments of the function overload that would be selected by a particular set of arguments. There are proposals that would help, and until that support is available, distinguishing the Invocable callback and input arguments must be done manually or by convention. There is also a need to supply the input arguments in one code path and the Invocable callback in another. Separating the Invocable callback argument out and supplying the Invocable callback arg later requires function binding, on top of the ability to distinguish the input and Invocable callback arguments (for now the options for this are manual and by-convention).

Taking the Invocable callback argument as the last argument affects valid signatures for and overloading of functions as well. Having a fixed last argument, requires adding overloads when defaulting the values of input arguments. Having a fixed last argument, requires that the overloads be constrainable so that the Invocable callback can be reliably distinguished from the input arguments. Having a fixed last argument, prevents using variadic input arguments, which is why `std::visit` places the Invocable callback as the first argument.

4.2.2 return value style

Using the return value from a function to separate the input and callback arguments allows the operation and the attachment of a callback to be deferred. Using a return value produces very different tradeoffs than when mixing callback and inputs together into the function arguments.

The Sender concept defined in [P1660] has been gaining support in SG1 recently. A signature for the `async_accept()` example might change to look something like:

```
template<
    typename SocketService
auto async_accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint);
```

When the callback is a return value, there is no need to be able to distinguish the input arguments from the callback argument. When the callback is a return value, the input arguments are the only arguments to the function and can be supplied in one code path and the callback to the return value of the function in another code path.

Returning a value that allows the callback to be attached later has no affect on valid signatures for and overloading of functions. Functions with overloads and variadic arguments behave no differently from any other function.

4.2.3 Networking TS style

The Networking TS [N4771] uses the completion tokens described in [N4045] to overload the callback argument to async functions like `async_accept()` (Accepting an Invocable callback indicates that the Networking TS style subsumes the function argument style and inherits the affects of having a fixed last argument.). On the other hand, if the argument is a completion token, then the completion token implements an Invocable callback that matches the completion signature for the function and delivers the result to that implementation. `async_accept()`'s signature, as defined in [N4771]:

```
template<
    typename SocketService,
    typename AcceptHandler>
void-or-deduced async_accept(
    basic_socket< protocol_type, SocketService > & peer,
    endpoint_type & peer_endpoint,
    AcceptHandler handler);
```

The completion token described in [N4045] and used in the Networking TS [N4771] has a limitation that was recently addressed in Boost.Asio. `async_result<>::initiate` (Boost.Asio revision history, github) was added recently to allow the completion token to describe a return value that would be allowed to defer the start of the async function and allow the callback to be attached later. The returned value would also be allowed to transfer the storage of the state for the operation to the caller, by storing that state in the returned value.

`async_initiate` enables the start of the operation to be Deferred, which can be used to reduce composition and runtime overhead. Without deferral it is complicated to write a generic retry function that starts an operation over again when it fails, because the async function has to be called with arguments and the callback/completion token.

`async_initiate` enables State transfer from the operation to the caller, which allows coroutine support that stores the state for the operation on the calling coroutine frame avoiding additional allocations.

The Networking TS [N4771] when updated to include `async_initiate` will allow a completion token to support deferral of the operation and separate the attachment of a callback from supplying the input arguments (`async_initiate` allows the creation of completion tokens that enables the Networking TS style to emulate the return value style). Emulating the return value style requires manually or by convention inserting the right completion token into the correct argument, which means that even when emulating the return value style the Networking TS style inherits the affects of having a fixed last argument from function argument style.

5 Proposals

The goal of this paper is to provide the rational for selecting One concept for functions taking callbacks and one concept for callbacks and the rational for the style on which each should be based.

5.1 Default Representation for new library callbacks

Of the Representations, the multiple function style is the one proposed by this paper, as the style to be adopted as a default style for all new callbacks in libraries for C++. The multiple function style was chosen even though the completion token style is already established as the callback mechanism in the Networking TS. The rational for this proposal follows. For the purpose of comparison this paper will use the Callback naming specified in [P1660] as an example of multiple function style. The names chosen for a particular expression of the multiple function style do not affect this proposal.

The data behind the rational is spread throughout the [Representations] section. To make the comparison easier the main points are represented in tables here:

Table 4: compare tradeoffs selected by completion token style with multiple function style

#	**completion token style**	**multiple function style**
1.	the callback is an **Invocable** or a completion token that **provides an Invocable** that will convert to some other style of callback	a callback is an object that is an **Invocable** and allows calls to `error()` and `done()` functions. Converting to other callback styles is a separate concern
2.	single functions will tend to be implemented as objects that have additional features like allocators and executors, though this implementation approach is not required. **simple functions** are allowed	callback **objects** will tend to be implemented by defining methods on an object, though this implementation approach is not required. Niebloid's will be used to allow free-function customizations for arbitrary types
3.	calls to the single function callback may have different execution guarantees that will depend on the **runtime value** of the parameters. One example is when an error may not be deliverable from the same execution context that is used to deliver the result	calls to **each function** on a callback may have different execution guarantees specified by the caller.
4.	argument types are **required** to represent invalid or empty states	argument types are **not required** to represent invalid or empty states
5.	the single function callback is **required** to add branches and checks for errors and validity before accessing the arguments	the functions are **not required** to add branches and checks for errors or validity
6.	all types are passed as **function arguments** with no required packing/unpacking	all types are passed as **function arguments** with no required packing/unpacking

#	completion token style	multiple function style
7.	a completion token **does not support overloads**, something like `std::variant` would be required to support multiple result types. a single function does support overloads.	each function **supports overloads** that allow different types to be supported without use of `std::variant`. this allows both error and value to have independent overloads
8.	a completion token **does not support overloads**, something like `std::optional` or `std::variant<std::tuple<>...>` would be required to support multiple result types. a single function does support overloads.	each function **supports overloads** that allow different numbers of arguments to be supported without use of `std::optional` or `std::variant<std::tuple<>...>`

5.2 Default Representation for new library functions taking callbacks

Of the Representations, the return value style is the one proposed by this paper, as the style to be adopted as a default style for all new functions taking callbacks in libraries for C++. The return value style was chosen even though the Networking TS style is already established. The rational for this proposal follows. For the purpose of comparison this paper will use the Sender naming specified in [P1660] as an example of return value style. The names chosen for a particular expression of the return value style do not affect this proposal.

The data behind the rational is spread throughout the [Representations] section. To make the comparison easier the main points are represented in tables here:

Table 5: compare tradeoffs selected by Networking TS style with return value style

#	Networking TS style	return value style
1.	adding the callback as the last argument makes overloading the function **more restrictive** than plain functions	returning a value that can attach the callback is **no more restrictive** than plain functions
2.	adding the callback as the last argument **prevents** variadic arguments to the function	returning a value that can attach the callback **does not prevent** variadic arguments to the function
3.	adding the callback as the last argument makes it **hard to distinguish** callback and non-callback arguments	returning a value that can attach the callback **clearly distinguishes** callback and non-callback arguments
4.	a particular completion token can implement `async_initiate` to **defer using the last argument and the return value**	a function can directly implement **defer using the return value**
5.	each completion token implementation is allowed to **alter the form** of all functions that use it	functions returning a value that can attach the callback **have a stable form** that makes it easier to write generic code

6 Usage for the proposed Representations

Table 6: compare coroutine usage for the existing Networking TS style & completion token style with the proposed return value style & multiple function style

Existing	Proposed

`socket_type t =`
`  co_await async_accept(`
`    socket,`
`    endpoint,`
`    use_coroutine);`
	`socket_type t =`
	`  co_await async_accept(`
	`    socket,`
	`    endpoint);`

Table 7: compare future & get usage for the existing Networking TS style & completion token style with the proposed return value style & multiple function style

Existing	Proposed

`socket_type t = async_accept(`
`  socket,`
`  endpoint,`
`  use_future).get();`
	`socket_type t = std::future_from(`
	`  async_accept(`
	`    socket,`
	`    endpoint)).get();`

Table 8: compare Invocable callback usage for the existing Networking TS style & completion token style with the proposed return value style & multiple function style

Existing	Proposed

`async_accept(`
`  socket,`
`  endpoint,`
`  [](`
`    error_code ec,`
`    socket_type s){});`
	`std::submit(async_accept(`
	`  socket,`
	`  endpoint),`
	`  std::callback(`
	`    [](error_code ec){},`
	`    [](socket_type s){}));`

7 Changes to the Standard

If this proposal is accepted then additional papers could add overloads to some existing functions that take callbacks. For example:

NOTE: There are cases, like the following, where usage of the new overloads is not as succinct as the current function definition. The motivation for adding these overloads would be to enable direct composition with algorithms and other functions taking callbacks.

Table 9: demonstrate potential new overload for `std::visit` that will make composition easier

Existing

```
std::variant<int, long, std::string> v{42};

std::visit(overloaded {
    [](auto arg) {},
    [](double arg) {},
    [](const std::string& arg) {},
}, v);
```

New

```
std::variant<int, long, std::string> v{42};

std::submit(
  std::visit(v),
  std::callback(
    overloaded {
      [](auto arg) {},
      [](double arg) {},
      [](const std::string& arg) {},
    }
  ));
```

Table 10: demonstrate potential new overload for `std::async` that will make composition easier

Existing

```
std::async(
  [](int i, const std::string& str){},
  42,
  "Hello");
```

New

```
std::submit(
  std::async(42, "Hello"),
  std::callback(
    [](int i, const std::string& str){}
  ));
```

8 Credits

This paper was influenced by hosts of people over decades.

— **Marc Barbour** and **Mark Lawrence** were fundamental to Kirk's first attempt to design more regular callbacks in a COM environment.
— **Aaron Lahman** was involved in that first attempt as well and introduced Kirk to the Reactive-Extensions libraries because he saw the similarity.

— **Erik Meijer** and his team took a very different path to arrive at a destination that resonated strongly with Kirk's goals
— *Microsoft Open Technologies Inc.* led by **Jean Paoli**, encouraged and supported Kirk's subsequent investment in finishing Aaron's C++ Rx prototype and then rewriting it to shift from interfaces to compile-time polymorphism.
— **Ben Christensen** drove changes to RxJava and his communication around those changes affected the design Kirk chose for rxcpp
— **Grigorii Chudnov**, **Valery Kopylov** and all the other amazing contributors to rxcpp over the years
— **Eric Niebler**, **Lee Howes** and **Lewis Baker** who more than anyone else contributed to the content of the motivation section of this paper
— *CppCon*, *CppNow*, *CppRussia* and *CERN* (and the people behind those including; **Jon Kalb**, **Bryce Adelstein-Lebach**, **Sergey Platonov**, **Axel Naumann**) for all the opportunities to communicate the vision for callbacks in C++
— **Gor Nishanov** for the excellent coroutines in C++20 and the shout-outs and support for rxcpp over the years.

9 References

[N4045] Christopher Kohlhoff. 2014. Library Foundations for Asynchronous Operations, Revision 2. https://wg21.link/n4045

[N4771] Jonathan Wakely. 2018. Working Draft, C++ Extensions for Networking. https://wg21.link/n4771

[P1055R0] Kirk Shoop, Eric Niebler, Lee Howes. 2018. A Modest Executor Proposal. https://wg21.link/p1055r0

[P1341R0] Lewis Baker. 2018. Unifying Asynchronous APIs in the Standard Library. https://wg21.link/p1341r0

[P1371R0] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. 2019. Pattern Matching. https://wg21.link/p1371r0