async-object - aka async-RAII

Proposal

Document #:	P2849R0
Date:	2024-05-21
Project:	Programming Language C++
Audience:	LEWG Library Evolution
Reply-to:	Kirk Shoop <kirk.shoop@gmail.com>

Contents

1 Introduction **1**
  1.1 What is an *async-object*? . . . 1

2 Example **2**
  2.1 The Foo *async-object* . . . 2
  2.2 `async_using` some *async-object* s . . . 3
  2.3 The `stop_object` *async-object* . . . 3
  2.4 Chaining *stop-source* s . . . 4

3 Motivation **5**

4 Design **6**
  4.1 What are the requirements for an *async-object*? . . . 6
  4.2 What is the concept that an *async-object* must satisfy? . . . 7
  4.3 What was the path to this design? . . . 14

5 Implementation Experience **16**

6 References **16**

1 Introduction

This paper describes concepts that would be used to create and cleanup an *async-object*, describes the `async_using` algorithm that will construct and destruct a pack of *async-object* s within an *async-function*, and describes the `async_tuple` type which is an *async-object* that composes the construction and destruction of multiple *async-object* s.

1.1 What is an *async-object*?

An *async-object* is an object with state that has *async-function* s to construct and destruct the object.

Examples include:

— thread
— thread-pool
— io-pool
— buffer-pool
— mutex
— file

— socket
— counting-scope

2 Example

[async-object POC (godbolt)]

2.1 The Foo *async-object*

Foo is a simple *async-object* that has an int for state.

```
// Foo is an async-object that stores an int
struct Foo {
  // defines an async-object
  // the object must be immovable
  struct object :  __immovable {
    object() = delete;

    int v;
  private:
    // only allow Foo to run the constructor
    friend struct Foo;
    explicit object(int v) noexcept : v(v) {}
  };
  // the handle type for the constructed object.
  // this is produced by async_construct
  using handle = std::reference_wrapper<object>;
  // the type that reserves storage for the object
  // must be nothrow default constructible and immovable
  using storage = std::optional<object>;

  // async_construct returns a sender that completes with a handle to the constructed object
  auto async_construct(storage& stg, int v) const noexcept {
    return  then( just(std::ref(stg), v),
      [](storage& stg, int v) noexcept {
        auto construct = [&]() noexcept { return object{v}; };
        stg.emplace( __conv{construct});
        printf("foo constructed, %d\n", stg.value().v);
        return std::ref(stg.value());
      });
  }

  // async_destruct returns a sender that performs the destruction the async-object
  // in storage and completes after the object is destructed
  auto async_destruct(storage& stg) const noexcept {
    return  then( just(std::ref(stg)),
      [](storage& stg) noexcept {
        printf("foo destructed %d\n", stg.value().v);
        stg.reset();
      });
  }
};

static_assert(async_object<Foo>);
```

```
static_assert(async_object_constructible_from<Foo, int>);
```

2.2 async_using some *async-object* s

This creates two Foo *async-object* s and modifies the state in a sender expression

```
int main() {
  // use async-objects
  auto inner = [](Foo::handle o0, Foo::handle o1) noexcept {
      return  then( just(o0, o1),
        [](Foo::handle o0, Foo::handle o1) noexcept {
          auto& ro0 = o0.get();
          auto& ro1 = o1.get();
          ro0.v = ro0.v + ro0.v;
          ro1.v = ro1.v + ro1.v;
          printf("foo pack usage, %d, %d\n", ro0.v, ro1.v);
          fflush(stdout);
          return ro0.v + ro1.v;
        });
    };

  // package async-object constructor arguments
  packaged_async_object foo7{Foo{}, 7};
  packaged_async_object foo12{Foo{}, 12};

  // reserve storage, construct async-osbjects,
  // give async-object handles to the given inner
  // async-function, and destruct async-objects
  // when the inner async-function completes
  auto use_s = async_using(inner, foo7, foo12);

  auto [v] =  sync_wait(use_s).value();
  printf("foo pack result %d\n\n", v);
}
```

2.3 The stop_object *async-object*

C++20 added `std::stop_source` and `std::stop_token` to represent a cancellation signal.

A `std::stop_source` object allocates state on the heap and shares it with the `std::stop_token` object.

[P2300R9] defines `inplace_stop_source` and `inplace_stop_token`. `inplace_stop_source` is an immovable object that contains the state inline. `inplace_stop_token` has a pointer to the state inside `inplace_stop_source`. This avoids an allocation, but the `inplace_stop_source` needs to be stored in an *operation-state* or some other stable location.

It is trivial to build an *async-object* around `inplace_stop_source` that can composed with other *async-object* s and *async-functions* s. This avoids the allocation of state and allows composition of multiple *async-object* s.

```
struct stop_object {
  using object = inplace_stop_source;
  class handle {
    object* source;
    friend struct stop_object;
    explicit handle(object& s) : source(&s) {}
```

```
  public:
    handle() = delete;
    handle(const handle&) = default;
    handle(handle&&) = default;
    handle& operator=(const handle&) = default;
    handle& operator=(handle&&) = default;

    inplace_stop_token get_token() const noexcept {
      return source->get_token();
    }
    bool stop_requested() const noexcept {
      return source->stop_requested();
    }
    static constexpr bool stop_possible() noexcept {
      return true;
    }
    bool request_stop() noexcept {
      return source->request_stop();
    }
    template<class Sender>
    auto chain(Sender&&);
  };
  using storage = std::optional<object>;

  auto async_construct(storage& stg) const noexcept {
    auto construct = [](storage& stg) noexcept -> handle {
      stg.emplace();
      return handle{stg.value()};
    };
    return  then( just(std::ref(stg)), construct);
  }
  auto async_destruct(storage& stg) const noexcept {
    auto destruct = [](storage& stg) noexcept {
      stg.reset();
    };
    return  then( just(std::ref(stg)), destruct);
  }
};
```

2.4 Chaining ***stop-source*** s

When a new stop source is created, there is a need to compose stop token from a receiver to stop the new stop source and to provide the new stop token to nested senders.

The `chain()` method on the `stop_object` handle does this composition.

This creates two new `stop_objects` and chains them together so that a stop request at any one of them will request all the nested stop sources to stop as well. `chain()` also ensures that `read_env()` will get the stop token for the innermost stop source.

```
enum class stop_test_result {stopped, unstopped};
void stop(stop_test_result sr) {
  auto inner = [sr](stop_object::handle source0, stop_object::handle source1) noexcept {
    auto body =  then( read_env( get_stop_token),
      [sr, source0](auto stp) mutable noexcept {
```

```
        // prove that chain() propagates from
        // env stop_token to stop source
        if (sr == stop_test_result::stopped) {
          // stop source0
          source0.request_stop();
        }
        // check source1
        return stp.stop_requested();
      });
    return source0.chain(source1.chain(body));
  };

  auto use_s = async_using(inner, stop_object{}, stop_object{});

  auto [stop_requested] =  sync_wait(use_s).value();
  printf("%s - %s\n\n",
    ((stop_requested && sr == stop_test_result::stopped) ||
     (!stop_requested && sr == stop_test_result::unstopped))
      ? "PASSED" : "FAILED",
    stop_requested ? "stop requested" : "stop not requested");
}
```

3 Motivation

It is becoming apparent that all the sender/receiver features are language features being implemented in library.

Sender/Receiver itself is the implementation of an *async-function*. An *async-function* can complete asynchronously with values, errors, or cancellation.

An *async-object* requires manual memory management in implementations because *async-object* s do not fit inside any single block in the language.

A major precept of [P2300R9] is structured concurrency. The `let_value()` algorithm provides stable storage for values produced by the input *async-function*.

It has been suggested that adding a `let_stop_source(sender, sender(inplace_stop_source&))` algorithm would allow a new `inplace_stop_source` to be created in a sender expression. Over time other bespoke `let_..` algorithms have been suggested to create other resources within sender expressions. This explosion of suggestions for bespoke `let_..` algorithms demonstrates that a general design for composing async resources is missing.

What is missing is a way to attach an *async-object* to a sender expression such that the *async-object* is constructed asynchronously before any nested *async-function* s start and is destructed asynchronously after all nested *async-function* s complete.

Asynchronous lifetimes in programs often use `std::shared_ptr` to implement ad-hoc garbage collection of objects used by asynchronous code. Using garbage collection for this purpose removes structure from the code, because the shared ownership allows objects to escape the original scope in which they were created.

Using objects like this in asynchronous code results in ad-hoc solutions for *async-construction* and *async-destruction*. No generic algorithms can compose multiple objects with asynchronous lifetimes when the *async-construction* and *async-destruction* are ad-hoc and follow no generic design.

The C++ language has a set of rules that are applied in a code-block to describe when construction and destruction of objects occur, and has rules that scope access to the successfully constructed objects within the code-block. The language implements those rules.

This paper describes how to implement rules for the construction and destruction of *async-object* s, using

sender/receiver, in the library. This paper describes structured construction and destruction of objects in terms of *async-function* s. The `async_using()` algorithm described in this paper is a library implementation of an async code-block containing one or more local variables. The `async_using()` algorithm is somewhat analogous to the `using` keyword in some languages.

4 Design

4.1 What are the requirements for an *async-object*?

async construction

Some objects have *async-function* s to run during construction that establish a connection, open a file, etc..

The design must allow for *async-function* s to be used during construction - without blocking any threads (C++ constructors are unable to meet this requirement)

async destruction

Some objects have *async-function* s to run during destruction that teardown a connection, flush a file, etc..

The design must allow for *async-function* s to be used during destruction - without blocking any threads (C++ destructors are unable to meet this requirement)

structured and correct-by-construction

These are derived from the rules for objects in the C++ language.

An *async-object* :

- — will not be available until *async-construction* has completed successfully
- — may complete *async-construction* with an error
- — may support cancellation of *async-construction*
- — must ensure that *async-destruction* will be no-fail and *unstoppable*

The `async_using` algorithm:

- — will always complete *async-construction* before invoking the inner *async-function*
- — will always complete *async-destruction* before completing to the containing *async-function*
- — will always invoke *async-destruction* when the inner *async-function* completes
- — will always invoke *async-destruction* of multiple *async-object* s in the reverse order of the *async-construction* of those *async-object* s
- — will always invoke *async-destruction* of any *async-object* s that successfully completed *async-construction*

The `async_tuple` type:

- — will always complete *async-construction* of all contained *async-object* s before completing the `async_construct` *async-function* of the `async_tuple`
- — will always invoke *async-destruction* of all contained *async-object* s in the reverse order of the *async-construction* of those *async-object* s
- — will always invoke *async-destruction* of any contained *async-object* s that successfully completed *async-construction*

composition

- — Multiple *async-object* s will be available at the same time without nesting.
- — Dependencies between objects will be expressed by nesting.
- — Composition will support concurrent *async-construction* of multiple objects.
- — Composition will support concurrent *async-destruction* of multiple objects.

4.2 What is the concept that an *async-object* must satisfy?

4.2.0.1 The `async_object` Concept:

An *async-object* provides the *async-function* s `async_construct` and `async_destruct` to construct an *async-object*.

An *async-object* provides the `object` type that contains the state for the constructed *async-object*. An `object` type must be immovable so that a moveable `handle` can refer to the constructed `object`. The constructors of the `object` type must only be available to the *async-object* implementation.

An *async-object* provides the `handle` type that is a ref-type that refers to the constructed `object`. A `handle` type must be moveable so that it can be passed into nested *async-function*. A `handle` type is not allowed to be empty, it is either moved-from and invalid to use or it refers to constructed `object`.

An *async-object* provides the `storage` type that contains storage for the `object`. A `storage` type must be default-constructible and immovable. This allows the `storage` to be reserved prior to construction of the `object` within and for the `handle` to refer to the constucted `object`.

A successfully constructed *async-object* :

- — is accessed through an *async-object-handle*
- — holds state and provides zero or more *async-functions* s that operate on the state *async-object*

```
/// @brief the async-object concept definition

template<class _T, class _O, class _H, class _S>
concept __async_object_members =
  std::is_move_constructible_v<_T> &&
  std::is_nothrow_move_constructible_v<_H> &&
  !std::is_default_constructible_v<_O> &&
  !std::is_move_constructible_v<_O> &&
  !std::is_constructible_v<_O, const _O&> &&
  std::is_nothrow_default_constructible_v<_S> &&
  !std::is_move_constructible_v<_S> &&
  !std::is_constructible_v<_S, const _S&>;

template<class _S>
concept __async_destruct_result_valid =
   __single_typed_sender<_S> &&
   sender_of<_S,  set_value_t()>;

template<class _T>
concept async_object =
  requires (){
    typename _T::object;
    typename _T::handle;
    typename _T::storage;
  } &&
  __async_object_members<_T,
    typename _T::object,
    typename _T::handle,
    typename _T::storage> &&
  requires (const _T& __t_clv, typename _T::storage& __s_lv){
    { async_destruct_t{}(__t_clv, __s_lv) }
      ->  __nofail_sender;
  } &&
  __async_destruct_result_valid<async_destruct_result_t<_T>>;
```

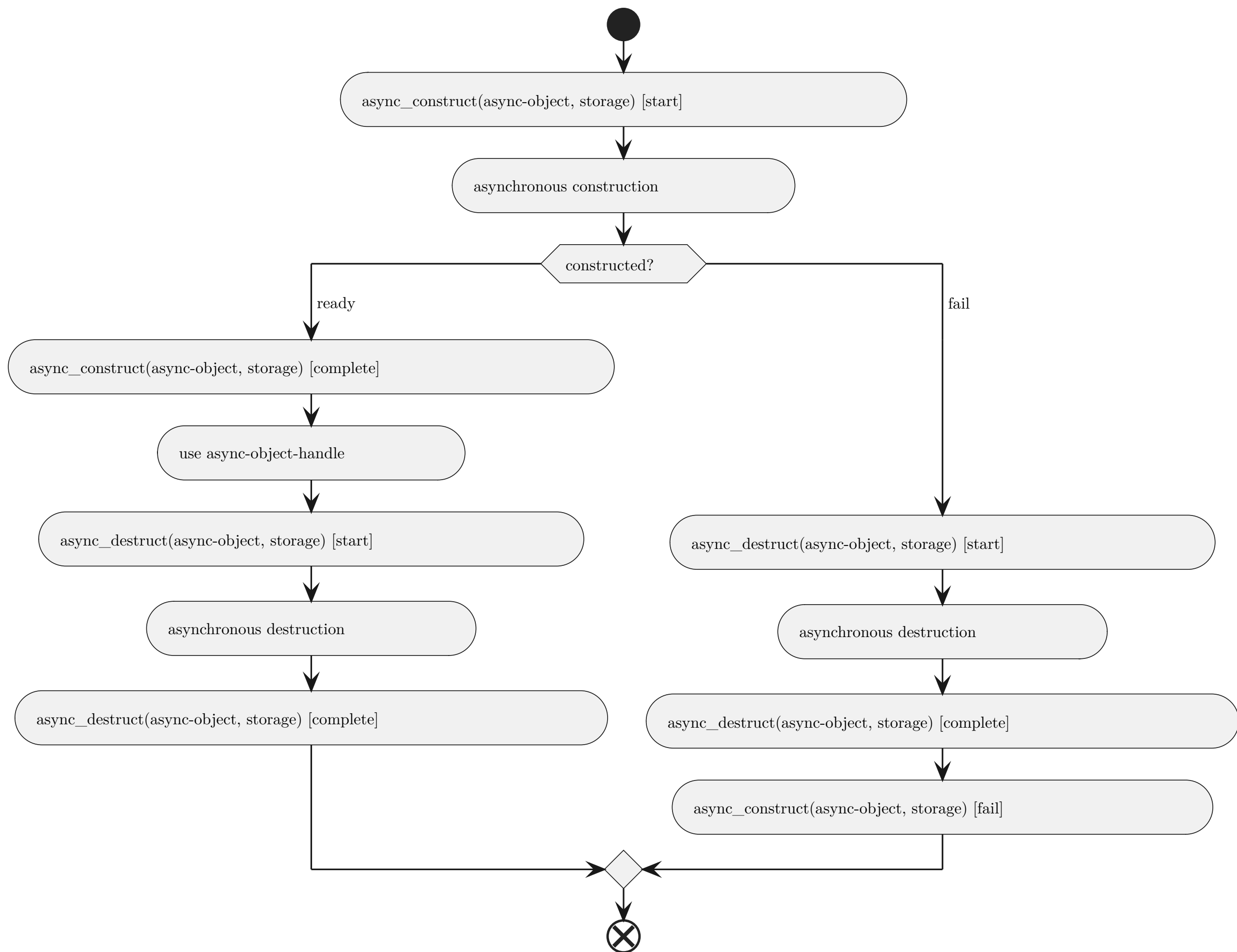
```
template<class _T, class... _An>
concept async_object_constructible_from =
  async_object<_T> &&
  requires (
    const _T& __t_clv,
    typename _T::storage& __s_lv, _An... __an){
    { async_construct_t{}(__t_clv, __s_lv, __an...) }
      ->  sender_of< set_value_t(typename _T::handle)>;
  };

using async_construct_t = /*implementation-defined/*;
/// @brief the async_construct() cpo provides a
///   sender that will complete with an async-object-handle.
/// @details The async-object-handle will be valid
///   until the sender provided by async_destruct()
///   is started.
/// @param obj A const l-value reference to an async-object
///   to be constructed
/// @param stg A mutable l-value reference to the
///   async-object-storage for the constructed async-object
/// @param an... The arguments to the async-object's async_construct()
/// @returns sender_of<set_value_t(async-object-handle)>
/**/
inline static constexpr async_construct_t async_construct{};

using async_destruct_t = /*implementation-defined/*;
/// @brief the async_destruct() cpo provides a
///   sender that will destroy the async-object
/// @details The async-object-handle becomes
///   invalid when the returned sender is started
/// @param obj A const l-value reference to an
///   async-object to be destructed
/// @param stg A mutable l-value reference to
///   the async-object-storage for the destructed async-object
/// @returns sender_of<set_value_t()>
/**/
inline static constexpr async_destruct_t async_destruct{};
```

4.2.0.2 Class diagram *async-object*

4.2.0.3 Activity diagram `async-object`



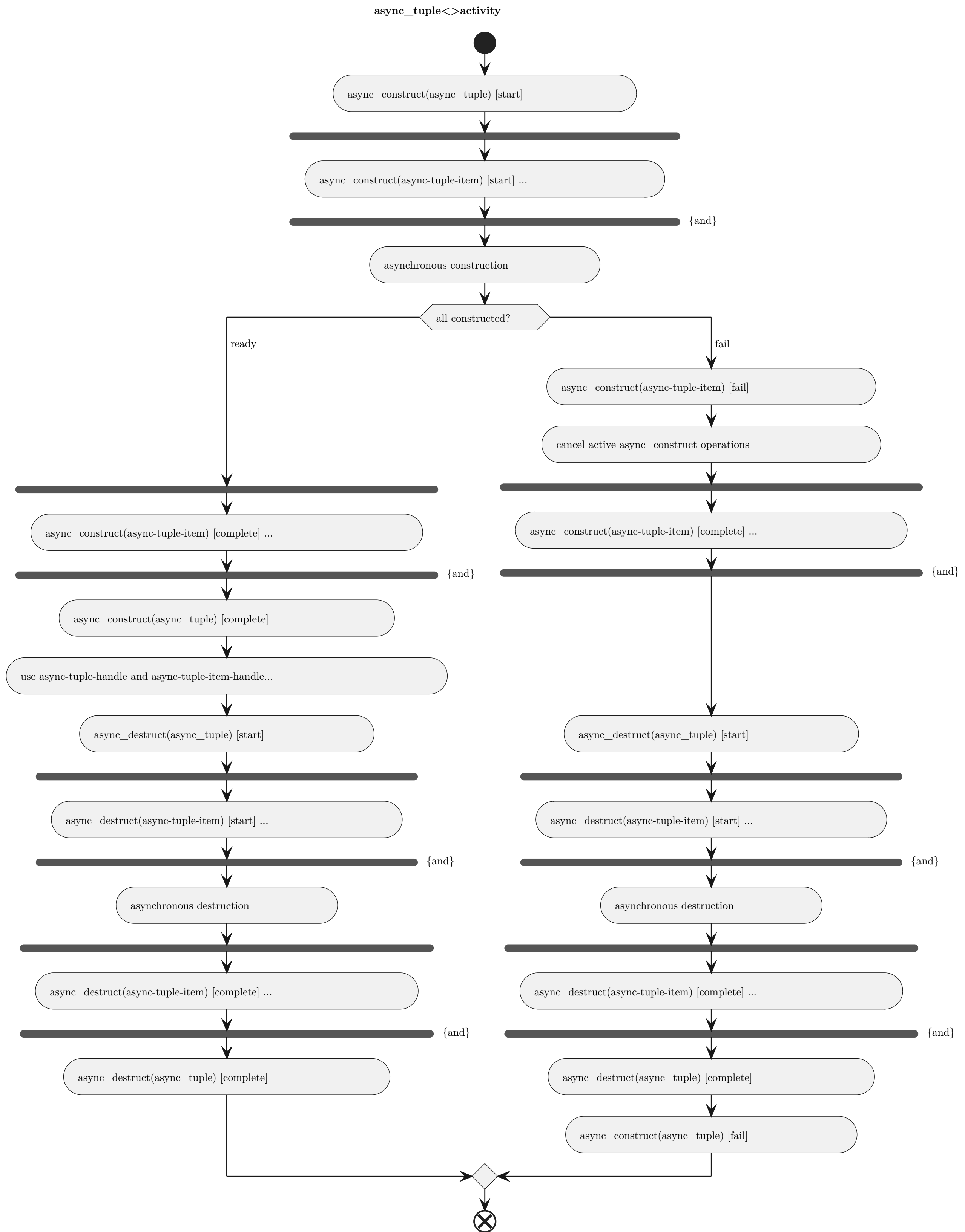
4.2.0.4 The `async_tuple<>` Type:

The `async_tuple<>` type is an *`async-object`* aggregates multiple *`async-object`* s.

An `async_tuple<>` can be used to place multiple *`async-object`* s members in a *`async-object`* without having to manually compose all the individual `async_construct` and `async_destruct`.

```
using make_async_tuple_t = /*implementation-defined/*;
/// @brief the make_async_tuple(aon...) cpo provides an
///   async_tuple<> that contains zero or more async-objects
/// @details The async_tuple<> is itself an async-object
///   that composes the async-construction and
///   async-destruction of the async-objects that
///   it contains
/// @param aon... a pack of packaged-async-objects.
/// @returns async_tuple<..>
/**/
inline static constexpr make_async_tuple_t make_async_tuple{};
```

4.2.0.4.1 Activity diagram `async_tuple<>`



4.2.0.5 The `async_using()` Algorithm:

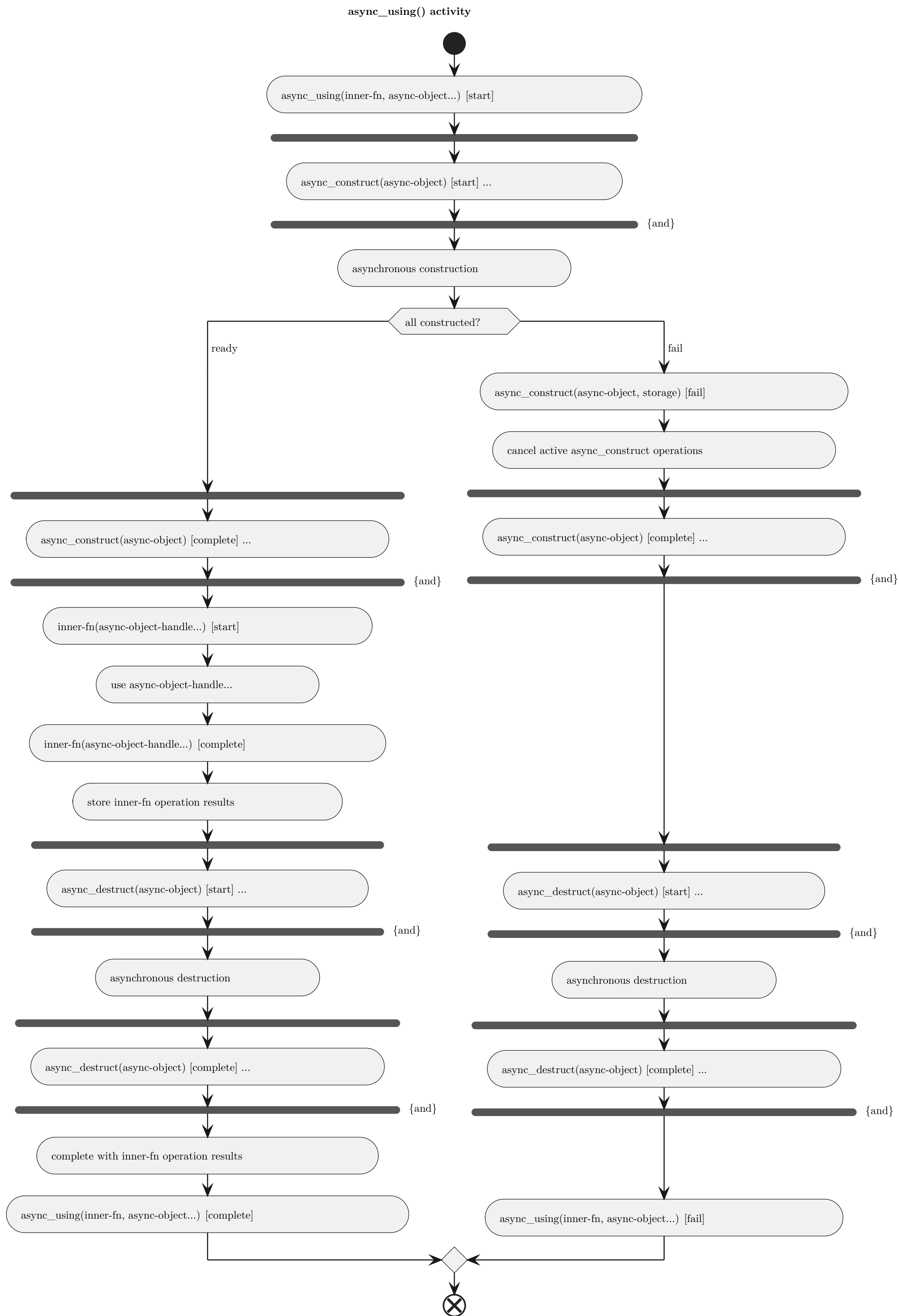
The `async_using()` algorithm is an *`async-function`* that opens a new function scope with one or more *`async-object`* handles available.

The `async_using()` operation will construct one or more *`async_object`* s and then all the *`async_object`* handles will be given to `inner-fn`. The `inner-fn` is an *`async-function`* that uses the handles. When the `inner-fn` operation completes, `async_using()` will destruct all the *`async-object`* s and then complete with the result of the `inner-fn` operation.

This is a library implementation of opening a block in a function with an open brace, where the compiler will construct local objects in the order that they are listed and destruct the locals in reverse order at the close brace.

```
using async_using_t = /*implementation-defined/*;
/// @brief the async_using(innerFn, aon...) cpo provides a
///   sender that completes with the result of the sender
///   returned from innerFn
/// @details The returned sender composes the construction
///   and destruction of the async-objects that it contains
///   and provides the async-object-handles to innerFn.
///   The innerFn returns an inner sender. The returned inner
///   sender is started.
///   When the returned inner sender
///   completes, the results are stored and async_destruct()
///   for each of the async-objects is started.
///   When all the async_destruct async-functions complete
///   then the returned sender completes with the stored
///   results of the inner sender.
/// @param innerFn A function that takes a pack of
///   async-object-handles and returns a sender.
/// @param aon... a pack of packaged-async-objects.
/// @returns async_tuple<..>
/**/
inline static constexpr async_using_t async_using{};
```

4.2.0.5.1 Activity diagram `async_using()`


async_using() activity
async_using(inner-fn, async-object...) [start]
async_construct(async-object) [start] ...
{and}
asynchronous construction
all constructed?
ready
fail
async_construct(async-object, storage) [fail]
cancel active async_construct operations
async_construct(async-object) [complete] ...
{and}
async_construct(async-object) [complete] ...
{and}
inner-fn(async-object-handle...) [start]
use async-object-handle...
inner-fn(async-object-handle...) [complete]
store inner-fn operation results
async_destruct(async-object) [start] ...
{and}
async_destruct(async-object) [start] ...
{and}
asynchronous destruction
asynchronous destruction
async_destruct(async-object) [complete] ...
{and}
async_destruct(async-object) [complete] ...
{and}
complete with inner-fn operation results
async_using(inner-fn, async-object...) [complete]
async_using(inner-fn, async-object...) [fail]

4.2.0.6 The `packaged_async_object<>` Type:

The `packaged_async_object<>` type is an *`async-object`* that is always default async-constructible. The `packaged_async_object<>` type stores a pack of arguments and in its defaulted `async_construct`, gives those arguments to the async_constructor of the nested *`async-object`*.

`async_using()` and `make_async_tuple()` require a pack of default async-constructible *`async-object`* s. The `packaged_async_object<>` type is used to make default async-constructible *`async-object`* s that can be passed to `async_using()` and `make_async_tuple()`.

```
using make_packaged_async_object_t = /*implementation-defined/*;
/// @brief the pack_async_object(o, an...) cpo provides a
///   packaged_async_object<> that stores arguments for the
///   async_construct() of the given async-object.
/// @details The packaged_async_object<O, An...> is itself an
///   async-object that composes the async-construction and
///   async-destruction of the given async-object.
///   The packaged_async_object provides an async_construct()
///   that takes no arguments and forwards the packaged arguments
///   to the given async-object async_constructor.
///   This is used to allow async_using() and async_tuple<> to
///   take a pack of async-objects that need no arguments.
/// @param o an async-object to package.
/// @param an... a pack of arguments to package.
/// @returns packaged_async_object<O, An...>
/**/
inline static constexpr make_packaged_async_object_t make_packaged_async_object{};
```

4.3 What was the path to this design?

Early designs were the result of seeing patterns in how async resources like *`async-scope`*, *`execution-context`*, *`stop-source`*, *`async-mutex`*, *`socket`*, *`file`*, etc..

All of the following designs were implemented and used.

4.3.0.1 `run()` & `join()`

An early design had `run()`and `join()`. This had composition issues and complexity issues.

4.3.0.2 `run()`, `open()`, and `close()`

This led to a design that had `run()`, `open()`, and `close()`. the `open()` and `close()` operations were nested inside the `run()` operation. The `open()` operation completed after the `run()` operation completed the async construction of the *`async-object`* and the `close()` operation completed after the `run()` operation completed the async destruction of the *`async-object`*.

This design was hard to communicate. The relationship between `open()` & `close()` and `run()` was confusing and took time to teach and understand. The implementation of an *`async-object`* with this design was very complicated.

4.3.0.3 *`async-sequence`*

This is hard to describe without covering *`async-sequence`* in detail.

The short form is that an *`async-sequence`* has an operation that delivers all the items in the sequence and each item in the sequence has an operation that processes that item.

There was a direct map of the `run()`, `open()`, and `close()` design to an *`async-sequence`* with one *`async-sequence-item`*. This meant that two separate designs - *`async-sequence`* and *`async-object`* - could be reduced to one design.

The *`async-sequence`* operation and the `run()` operation were a direct map and the single *`async-sequence-item`* operation was a direct map to the `open()` operation (the handle to the constructed object that the `open()` operation completed with became the single *`async-sequence-item`*).

The *`async-sequence`* operation constructed the *`async-object`* and then emitted the handle to the constructed *`async-object`* as the single *`async-sequence-item`*, and once the operation that processed the *`async-sequence-item`* completed, the *`async-sequence`* operation would destruct the object, and finally the *`async-sequence`* operation would complete.

This provided users with correct-by-construction usage. The implementation of *`async-object`* s with this design was complex.

4.3.0.4 `async_construct()`, `async_destruct()`, `object`, `handle`, and `storage`

This design followed from feedback that the previous designs had semantics that were difficult to explain and required *`async-object`* implementations to be very complex and placed constraints on the usage that limited the ways in which *`async-object`* s could be composed.

The feedback suggested that the object should be concerned with construction and destruction and not with how those are composed during usage. It was recommended that the design of objects in the language with independent functions to construct and destruct was the model to follow.

The result is that an implementation of an *`async_object`* is not very complex. The complexity is in the `async_using()` and `make_async_tuple()` implementations.

Most of the previous designs had something like `async_using()`, but this is the first implementation of `async_using()` that supported a pack of objects. This is also the first time that the `async_tuple<>` type was implemented to compose multiple *`async-object`* s as members of a containing *`async-object`*.

The success in implementing the composition of *`async-object`* s to support these different use cases is an encouraging confirmation of this design.

4.3.0.4.1 variation:

There has been discussion about another way to achieve the ability for `async_using()` and `make_async_tuple()` to take a pack of *`async-object`* s to compose.

The *`async-object`* would drop the `storage` member type. The *`async-object`* would be immovable and default-constructible (it would become the `storage` type). The `async_construct()` and `async_destruct()` functions would remove the `storage&` arguments and use `this` to access the storage.

In this design, the `packaged_async_object` would not be an *`async-object`*. An *`async-object`* would be immovable, but `packaged_async_object` must be moveable since it stores the pack of `async_construct()` arguments and must be passed as part of the pack given to `async_using()` and `make_async_tupe()`. The `packaged_async_object` type would almost be an *`async_object`* in this design. It would have `async_construct()` and `async_destruct()` members. It would need a `storage` type member that is the actual *`async-object`*. usage would require that the caller reserve space for the `storage` and pass the `storage` ref to `async_construct()` and `async_destruct()`.

This variation would require two concepts that have a lot of overlap.

- *`async-object`* would be immoveable and have a `handle` type member and `async_construct()` and `async_destruct()` members.
- *`packaged-async-object`* would be moveable and have a `storage` type member and `async_construct()` and `async_destruct()` members that take a reference to the `storage`.

The design selected in this paper chooses to have one concept that can be satisfied by all *`async-object`* s, including `packaged_async_object`.

5 Implementation Experience

`async-object` s have been built multiple times with different designs over several years. A lot of feedback has been collected from those other implementations to arrive at this design.

The design in this paper is implemented, but is not used widely at this time (May 2024).

The design in this paper has withstood the first round of feedback and is ready for a wider audience.

6 References

[async-object POC (godbolt)] async-object POC (godbolt).
https://godbolt.org/z/rrbW6veYd

[P2300R9] Eric Niebler, Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Bryce Adelstein Lelbach. 2024-04-02. 'std::execution'.
https://wg21.link/p2300r9