Paper no.	P1342R0
Date	2018-11-19
Reply To	Lewis Baker <lbaker@fb.com>
Audience	Evolution

Unifying Coroutines TS and Core Coroutines

Abstract

There is a desire to ship coroutines support in the C++20 time-frame and there is a mature and well-tested TS offered for this purpose. However, there are some concerns over the TS which have led to an alternative proposal (Core Coroutines). This paper offers a unification route and a path forward.

One of the concerns raised about the design of the Coroutines TS is that calling a coroutine can result in an implicit heap allocation. While the compiler can and often does elide the allocation, the language does not provide a guarantee that the allocation will be elided.

The need for a memory allocation is a result of the concrete coroutine frame not having a first-class type that is directly available to the developer. The compiler type-erases the coroutine frame and uses a fixed algorithm controlled via a number of library customization points to allocate the coroutine frame.

The Core Coroutines proposal described in P1063 seeks to address this issue by providing direct access to the coroutine frame object in the C++ type system, allowing the caller to control the placement of the coroutine state more directly. The coroutine transformation algorithm is more flexible and controlled via a smaller set of customization points than that of Coroutines TS.

However, the Core Coroutines approach as described in P1063R1 has some fundamental limitations with regards to the ability to perform tail-recursive resumption of coroutines across type-erased or ABI boundaries – a case that will be common in large code-bases. There are also some use-cases currently served by the Coroutines TS that are, as far as the author is aware, not yet possible to implement under the current Core Coroutines design.

While it's likely that the Core Coroutines design can continue to be iterated on to address these issues, it might take a long time to get there and it's not clear that the end-result would be significantly simpler than the Coroutines TS.

We recommend that, rather than discarding the tested and proven design of the Coroutines TS and starting from a clean slate, a refinement of the Coroutines TS design be offered that incorporates some of the design features from Core Coroutines to allow code to create coroutine types that are guaranteed not to be heap allocated.

The combined design also retains the ability for libraries to take advantage of compiler-generated type-erasure and allocation of the coroutine frame in cases where type-erasure is required (eg. across virtual calls or ABI boundaries). This allows the compiler to optimize coroutine frame sizes and inline allocations where possible – the latter is something the compiler cannot easily do when the type-erasure and allocation are introduced at the library level.

Finally, this paper proposes a path that allows us to ship support for part of this design in C++20, the part allowing type-erased coroutines as currently supported by the Coroutines TS, while leaving the door open to generalising coroutines to also give the library direct access to the coroutine object in the C++ type system in a future version of the C++ standard.

Motivation

The Coroutines TS has been implemented and usable within two major compilers for several years now and has a large amount of usage experience behind it. The concept of awaitable types and coroutines has been shown to be both powerful and flexible with regards to implementing abstractions for asynchronous programming.

However, the paper P0973R0 pointed out several issues with the Coroutines TS. The #1 issue identified there was that of the implicit memory allocation required for the coroutine frame. While the compiler is allowed to, and in many cases can, optimize the coroutine so that the memory allocation is elided, there is currently no easily defined wording we can put in the specification that would allow us to mandate that the heap allocation *must* be elided in certain cases.

Programmers will therefore forever be suspicious and constantly wonder whether the compiler has been able to optimize out the allocation in particular cases. A seemingly innocuous change may confuse the optimizer and accidentally introduce a performance regression.

The “Core Coroutines” paper P1063 seeks to address this issue by introducing the concept of a coroutine lambda that allows the caller to produce a coroutine frame object as a first-class type in the type system. This would allow code to place the coroutine frame object on the stack, if desired, and thus statically guarantee that there would be no heap allocation. This is particularly important for non-async coroutine usages, such as those returning a std::optional or std::expected value, where the cost of an extra memory allocation could be a significant performance regression.

However, I believe there are fundamental tradeoffs being made by making the coroutine frame a first-class type in the type-system compared with letting the compiler type-erase the coroutine frame:

If the coroutine frame is represented as a type in the type system then:

- We know the size of the coroutine frame at compile time and thus can reserve memory for it on the stack or inside other objects and thus guarantee that there are no heap allocations.
- The size of this coroutine frame needs to be known early in the compilation, before inlining, dead-code elimination and other optimisations are performed. This means that the compiler must be conservative in determining which code-paths are reachable and which variables have lifetimes that span suspend-points and thus need to be stored in the coroutine frame.
- This means the size of the coroutine frame will often be larger than is necessary.
- When the library does need to heap-allocate and/or type-erase the coroutine frame then the compiler will have greater difficulty in eliding the allocation or devirtualising calls to the coroutine compared with heap-allocation and type-erasure that was added by the compiler itself.
- As the coroutine frame size is fixed, the compiler does not have the flexibility to increase the size of the coroutine frame to inline/elide heap allocations of objects whose lifetimes are strictly nested within the coroutine.

If the coroutine frame is type-erased and heap-allocated by the compiler then:

- The size and layout of the coroutine frame can be calculated after optimization passes have been performed.
- This can lead to smaller coroutine frame sizes and thus lower memory utilization in the case that the compiler was able to eliminate dead-code or suspend-points after inlining.
- It can also allow the compiler to increase the coroutine frame size if required to inline the allocation of nested coroutine frames into the frame of the caller. This in turn can lead to fewer overall heap allocations when type-erased coroutines call other type-erased coroutines.
- The type-erasure added by the compiler is intrinsically known-about by the compiler and so can more easily be devirtualised/removed compared with type-erasure added at the library level.
- Memory allocations can fail due to memory exhaustion. This means there is an extra failure mode that needs to be handled by applications that call dynamically allocated coroutine frames. It becomes difficult to mark coroutine functions as noexcept.
- The programmer cannot guarantee that any particular call to a coroutine will have its coroutine frame allocation elided by the compiler. This can make it difficult to reason about performance of the code.

Core Coroutines has been targeting the first set of tradeoffs while the Coroutines TS has been targeting the second.

Ideally, we want a design for C++ coroutines that supports both approaches and that allows the library writer to make the choice between these tradeoffs based on their use-case.

Design Alternatives

We propose merging Coroutine TS as is, as we believe it is sufficient to allow adding support for Core Coroutine like transformation in C++Next in non-breaking fashion. We also, suggest a number of simplifications we can add in C++20 timeframe which, although not strictly necessary, reduce the customization surface area.

Incremental Path: No change to the Coroutines TS

In this design alternative, the Coroutines TS design is left intact and ships as-is with C++20 (possibly modulo the simplifications suggested in the next section). Support for Core Coroutines can be added to C++Next by making the compiler's handling of a coroutine conditional on the types and functions found within the coroutine's `coroutine_traits` specialization. If `coroutine_traits` contains a nested type named `promise_type`, then the compiler uses Coroutines TS-style dynamic frame allocation and type-erasure. If `coroutine_traits` has a static member function `get_return_object()`, the compiler uses Core Coroutines-style static frame allocation without type erasure. (See the Unified Coroutines design described below for how such a `get_return_object()` function can support the Core Coroutines use case.)

This is an example of a change in wording we envision in C++Next to enable selection of support of Core Coroutines:

Modify [dcl.fct.def.coroutine]/3 as follows:

> Let R be the return type and F be the function-body of f, T be the type std::experimental::coroutine_traits<R,P1,...,Pn>, and if the qualified-id T::promise_type is valid and denotes a type(17.9.2) ~~P be the class type denoted by T::promise_type~~. Then, the coroutine behaves as if …: otherwise, if the qualified-id T::get_return_object is valid and denotes a static member function, then the coroutine behaves as if …

The following wording assumes some of the changes to Core Coroutines suggested later in the Unified Coroutine section.

This approach to unification causes the least disruption to the existing design: it is possible to get many of the benefits of Core Coroutines in a later release of C++ after delivering TS-style coroutines in C++20.

In the next section we suggest some simplifications to the Coroutines TS design that loses no functionality. In addition to being desirable in their own right, these simplifications are a step toward a deeper unification of the TS with Core Coroutines, the vision of which is presented in a later section – a vision that likely doesn't fit in C++20, but that can be achieved iteratively over time.

Small Simplifications to the Coroutines TS for C++20

Here is a summary of the suggested simplifications. Later sections expand on the motivation and reasoning behind the changes.

Remove initial_suspend

Remove the initial_suspend point and always create the coroutine suspended and pass a handle of a suspended coroutine as an argument to get_return_object. For use cases where coroutine needs to start execution immediately, get_return_object can call resume() on the passed in coroutine handle as needed.

Simplify final_suspend

Coroutine now always suspends at the final suspend point. final_suspend customization is retained but now accepts the coroutine_handle for the current coroutine and returns a coroutine_handle to symmetrically resume or noop_coroutine handle if execution should be transferred to the caller of resume(). To implement fire-and-forget coroutines, the library writer can explicitly destroy the coroutine from final_suspend.

Rename await_transform

Rename await_transform to await_value and make it required for any coroutine that has a co_await in its body. This creates a nice symmetry: three co_ keywords (co_await, co_yield, and co_return), and three customization points (await_value, yield_value, and return_value); and each customization point is needed if the keyword is used.

Make unhandled_exception optional

The unhandled_exception customization point is optional. If a promise type does not define unhandled_exception, then the body of the coroutine is not wrapped in a try/catch, and the exception propagates out of coroutine_handle.resume(). This is now defined behavior with the adoption of the resolution of Coroutine TS issue 25. In addition to making it simpler to define synchronous coroutine type, it also greatly helps code generation in that case.

Simplify await_suspend

Have only coroutine_handle-returning await_suspend. bool- and void-returning variants of await_suspend can be expressed in terms of the former with use of noop_coroutine().

Although this isn't the radical simplification that Core Coroutines is aiming for, these are logical incremental improvements to a well-understood design that reduce the boilerplate necessary to author coroutine types and lighten the teaching burden.

A Unified Design

The final design alternative explores comprehensive unification of Coroutines TS and Core Coroutines. It retains the ability of the compiler to perform the allocation and type-erasure

from the Coroutines TS while also incorporating the ability to guarantee zero heap-allocations that Core Coroutines enabled.

This builds upon the mature design of the Coroutines TS, adding extra facilities, eliminating previously hard-coded algorithms and also simplifying the design.

Coroutine Objects

When a user calls a coroutine function, the first thing the compiler does is capture the arguments in a *coroutine object.*

The coroutine object is much like a lambda object in that it contains captures and has a compiler-generated, anonymous type. However, in this case it is the parameters of the coroutine that are captured rather than local variables from the surrounding scope.

The type and value category of the captures are by default the same as the type and value category of the corresponding parameter.

A *coroutine object* contains the captured variables and represents a factory for creating a coroutine frame:

```
// An archetypal coroutine object
template<typename... Captures>
struct Coroutine
{
  // Coroutine object is copy-constructible if all captures are
  // copy-constructible.
  Coroutine(const Coroutine&)
    noexcept(std::is_nothrow_copy_constructible_v<Captures> && ...)
    requires CopyConstructible<Captures> && ...;

  // Coroutine object is move-constructible if all captures are move
  // constructible.
  Coroutine(Coroutine&&)
    noexcept(std::is_nothrow_move_constructible_v<Captures> && ...)
    requires MoveConstructible<Captures> && ...;

  // Method for creating a coroutine-frame with statically known type.
  // Specifies the promise-type to use and constructs the promise-object
  // in-place in the coroutine frame.
  //
  // Returns the coroutine frame object by value with guaranteed copy
  // elision. The returned object is not movable so the returned object
  // must be placed in its final location when this is called.
  //
  // See below for a description of the interface of the coroutine-frame
  // object.
  //
  // This overload moves the captured parameters from the coroutine_object
  // into the coroutine frame object.
  template<typename Promise, typename... Args>
  auto create_static(Args&&... promiseArgs) && -> CoroutineFrameObject;
```

```
  // This overload creates a coroutine frame that captures a reference
  // to this Coroutine object and references the parameters in there
  // rather than moving the parameters into the frame.
  template<typename Promise, typename... Args>
  auto create_static(Args&&... promiseArgs) & -> CoroutineFrameObject;


  // Create the coroutine frame using dynamic allocation.
  //
  // Caller passes an allocator object which is used to allocate memory
  // for the coroutine frame if the compiler requires it. The compiler is
  // allowed to elide the call to the allocator if it does not require
  // additional memory (eg. if the coroutine frame memory can be inlined
  // into the stack/coroutine frame of the caller).
  //
  // Returns a handle to the dynamically-allocated coroutine frame.
  // The return type is convertible to the coroutine_handle<Promise>.
  //
  // Vague here to allow for a future extension to one day return a
  // per-suspend-point strongly-typed coroutine_handle.
  template<typename Promise, typename Allocator, typename... Args>
  auto create_dynamic(Allocator&& allocator, Args&&... promiseArgs) &&
    -> ConvertibleTo<coroutine_handle<Promise>>;

  // Same as above - does not copy captured values into coroutine frame.
  template<typename Promise, typename Allocator, typename... Args>
  auto create_dynamic(Allocator&& allocator, Args&&... promiseArgs) &
    -> ConvertibleTo<coroutine_handle<Promise>>;

  // Provide access to the captured parameters
  // This would be useful for some use-cases where the wrapper type
  // wants to be able to peek at the parameters
  // eg. the implicit object parameter
  // Could be added later if needed.
  //
  // Could return some unspecified tuple-like type instead of std::tuple
  // eg. a struct that have access to the named captured members.
  std::tuple<Captures&...> get_captures() &;
  std::tuple<Captures&&...> get_captures() &&;

};
```

Coroutine Frame objects

Then if we look at the interface for a CoroutineFrameObject returned by `create_static<Promise>()`

```
// Helper for conditionally defining .resume() as noexcept.
template<typename Promise>
constexpr bool __has_noexcept_unhandled_exception_v = false;

template<typename Promise>
  requires requires(Promise& promise)
  {
    { promise.unhandled_exception() } noexcept -> void;
```

```
  }
constexpr bool __has_noexcept_unhandled_exception_v = true;


template<typename Promise>
struct CoroutineFrameObject
{
  // Destroys the coroutine and its state. Runs destructors of all
  // captured variables, in-scope local variables, promise object.
  // Only valid if the coroutine is currently suspended.
  ~CoroutineFrameObject();

  // Coroutine frame is not copyable or movable.
  CoroutineFrameObject(CoroutineFrameObject&&) = delete;
  CoroutineFrameObject(const CoroutineFrameObject&) = delete;
  CoroutineFrameObject& operator=(const CoroutineFrameObject&) = delete;
  CoroutineFrameObject& operator=(CoroutineFrameObject&&) = delete;

  // Resume execution of the coroutine.
  // Not guaranteed to participate in tail-calls.
  void resume() noexcept(__has_noexcept_unhandled_exception_v<Promise>);

  // Query if the coroutine is currently suspended at the final
  // suspend-point.
  bool done() const;

  // Get a type-erased handle to this coroutine frame.
  coroutine_handle<Promise> get_handle();
  coroutine_handle<const Promise> get_handle() const;

  // Get access to the promise object
  Promise& promise() noexcept;
  const Promise& promise() const noexcept;
};
```

Storing promise externally from coroutine frame

If you don't want the promise object to be owned by the coroutine frame, you can specify the `Promise` template argument as `Promise&` and pass an lvalue reference to the promise object as the promise constructor argument.

eg.

```
my_promise promise;
auto frame = coroutine.create_static<my_promise&>(promise);
assert(&frame.promise() == &promise);
```

Compiler-provided type erasure with `coroutine_handle<T>`

The interface for `coroutine_handle` is as follows (this is mostly the same as Coroutines TS).

Note the similarities between the `coroutine_handle` interface and the `CoroutineFrameObject`. A `coroutine_handle` is simply a type-erased pointer/handle to a `CoroutineFrameObject`.

```
template<typename Promise = void>
class coroutine_handle;

template<>
class coroutine_handle<void>
{
public:
  constexpr coroutine_handle() noexcept; // null handle
  constexpr coroutine_handle(const coroutine_handle&) noexcept;
  constexpr coroutine_handle& operator=(const coroutine_handle&) noexcept;

  static coroutine_handle from_address(void* ptr);
  void* address() const noexcept;

  // Query for nullness of the coroutine handle.
  constexpr explicit operator bool() const noexcept;
  constexpr bool operator!() const noexcept;

  // Methods below are only valid to call if coroutine handle is non-null
  // and the coroutine is currently suspended.

  bool done() noexcept const;
  void resume() const;
  void destroy() const;

private:
  void* ptr; // exposition only
};

bool operator==(coroutine_handle<void> a, coroutine_handle<void> b);
bool operator!=(coroutine_handle<void> a, coroutine_handle<void> b);

template<typename Promise>
class coroutine_handle : public coroutine_handle<void>
{
public:
  coroutine_handle() noexcept; // null handle
  coroutine_handle(const coroutine_handle&) noexcept;
  coroutine_handle& operator=(const coroutine_handle&) noexcept;

  static coroutine_handle from_address(void* ptr);

  Promise& promise() const noexcept;

  void resume() const
    noexcept(__has_noexcept_unhandled_exception_v<Promise>);
};
```

NOTE: I have removed `coroutine_handle<Promise>::from_promise()` as I believe it to no longer be needed. Library code now has other more direct ways to get hold of the coroutine handle.

The Promise interface

The interface for a Promise type must have the following shape:

```
struct Promise
{
  // Mandatory. Executed when the coroutine runs to completion.
  // Returns the coroutine_handle of the continuation to transfer
  // execution to.
  // It is passed a handle to the coroutine that has just been suspended.
  // This allows the implementation to call h.destroy() if desired.
  coroutine_handle<T> final_suspend(coroutine_handle<Promise> h) noexcept;

  // Optional - only needed if promise wants to handle exceptions
  // If this is not defined then unhandled exceptions will propagate out
  // of calls to coroutine_handle::resume().
  void unhandled_exception();

  // Optional - opt-in to supporting implicit or explicit 'co_return;'.
  void return_void();

  // Optional - opt-in to supporting 'co_return value;'.
  template<typename T>
  void return_value(T&& value);

  // Optional - opt-in to supporting 'co_await value'.
  template<typename T>
  Awaitable await_value(T&& value);

  // Optional - opt-in to supporting 'co_yield value'.
  template<typename T>
  Awaitable yield_value(T&& value);
};
```

NOTE: As this proposal would require changing the interface of the `promise_type` from Coroutines TS in a non-backwards compatible way, I have taken the opportunity to rename `await_transform()` to `await_value()` and made it mandatory for a promise to implement in order to support the '`co_await`' operation. This change is not strictly necessary, but I feel helps to give the design more symmetry and also helps to simplify some of the complexity in deducing the result-type of a `co_await` expression given a `Promise` type.

NOTE: Possible future extension is to allow `return_void()` and `return_value()` to return `Awaitable` type to allow `co_return` statements to be suspend-points too. This could allow optimizations that eliminate storing a copy of the value in the promise. The promise could capture a pointer to the return value and then suspend, then `await_resume()` could copy it directly. Note that we need to allow suspend-points in catch-blocks before this can be supported and so this feature is deferred for now until more research can be done into the implications of suspending a coroutine within catch-blocks.

Noop Coroutine

As we want to be able to allow a coroutine to optionally specify that its continuation should be the caller of `.resume()` rather than another coroutine, we need to provide a special compiler/runtime-provided `coroutine_handle` that specifies the continuation should be to return from the call to `.resume()`. In Coroutines TS this was called `noop_coroutine()` and we adopt that terminology here.

```
struct noop_coroutine_promise {};

template<>
class coroutine_handle<noop_coroutine_promise>
  : public coroutine_handle<void>
{
  constexpr bool done() const noexcept { return false; }
  constexpr void resume() const noexcept {}
  constexpr void destroy() const noexcept {}
};

constexpr coroutine_handle<noop_coroutine_promise> noop_coroutine();
```

NOTE: It may be worth bikeshedding the name of this thing to more accurately represent what it represents – the continuation is the caller of `.resume()` rather than another coroutine.

Dynamic allocation of coroutine frames

When a program calls the `Coroutine::create_dynamic()` interface the semantics of this is defined as follows.

```
struct Coroutine
{
private:
  // Unspecified contents holding captured variables.
  std::tuple<Captures...> captures;

  // Potential definition of compiler-generated hidden coroutine frame
  // type - exposition only.
  template<typename Allocator, typename Promise, typename CapturePack>
  struct CoroutineFrame
  {
    template<typename... Args>
    CoroutineFrame(Allocator allocator, CapturePack&& capturePack,
                   Args&&... args)
    : __allocator(std::move(allocator))
    , __promise(std::forward<Args>(args)...)
    , __captures(std::forward<CapturePack>(captures))
    {
      // unspecified other initialization of coroutine frame.
    }

    ~CoroutineFrame()
    {
      // Destroys any variables currently in-scope.
    }
```

```
    Promise& promise() { return __promise; }
    const Promise& promise() const { return __promise; }

    coroutine_handle<Promise> get_handle() { ... }

    void resume() { ... }

    void destroy()
    {
      auto allocator = std::move(__allocator);
      this->~CoroutineFrame();
      allocator.deallocate(this, sizeof(CoroutineFrame));
    }

  private:
    // Unspecified, compiler generated layout (exposition only)
    Allocator __allocator;
    Promise __promise;
    Captures __captures;
    char __storage[unspecified];
  };

public:

  template<typename Promise, typename Allocator, typename... Args>
  auto create_dynamic(Allocator alloc, Args&&... args) &&
  {
    using char_allocator_t =
      std::allocator_traits<Allocator>::rebind_alloc<char>;
    using coroutine_frame_type =
      CoroutineFrame<char_allocator_t, Promise, std::tuple<Captures...>>;

    char_allocator_t charAlloc{ alloc };

    // TODO: How do we specify alignment requirements to an allocator?

    void* buffer = charAlloc.allocate(sizeof(coroutine_frame_type));
    try
    {
      auto* frame = ::new (buffer) coroutine_frame_type{
        charAlloc, std::move(this->captures), std::forward<Args>(args)...};
      return frame->get_handle();
    }
    catch (...)
    {
      charAlloc.deallocate(buffer, sizeof(CoroutineFrame));
      throw;
    }
  }

  // The overload of create_dynamc() that is lvalue qualified differs only
  // in that the 'Captures' template parameter is an lvalue reference to
  // the captures stored in this Coroutine object instead of
  // moving-constructing copies of the parameters into the frame.
  // Calling this overload would allow the coroutine wrapper to avoid an
  // extra copy of the parameters.
```

```
};
```

The main thing to note here is that the compiler can defer calculation of `sizeof(CoroutineFrame)` until after it has performed optimization passes: inlining, heap elision, etc. But this should give the rough semantics of the heap allocation in terms of the allocator object passed to the `create_dynamic()` function.

Sugar Syntax

We now have the core building blocks for creating coroutines: coroutine objects, coroutine frame objects and compiler-aware type-erased coroutine_handles.

The next step is to provide a mechanism to construct the coroutine object and expose it to library code.

What we would like to write is the following (as we can with Coroutines TS) and have this translated by the compiler into something which the library can customise easily in terms of the coroutine object:

```
task<int> foo(int a)
{
  int b = co_await bar(a);
  int c = co_await baz(b);
  co_return c;
}
```

For the purposes of exposition, let's assume the existence of a coroutine-lambda syntax for constructing a coroutine object. We borrow the syntax proposed in the Core Coroutines proposal for describing a coroutine lambda expression, although we do not necessarily need to add such syntax to the language at this point in time.

The general syntax of the expositional coroutine lambda expression is:

```
[<capture-list_opt>] [->] { <coroutine-body> }
```

Where the *capture-list* is the same as for normal lambdas. The result of evaluating the lambda expression is a new *CoroutineObject* prvalue.

The primary customization point for this is the `coroutine_traits` type-trait that was defined by the Coroutines TS. However, we modify it to allow customization of the `get_return_object()` function instead of customization of the `promise_type`. This allows the library to implement the algorithm used to construct a coroutine frame and a return-value object.

```
template<typename Ret, typename... Args>
struct coroutine_traits
{
  template<typename Coroutine>
```

```
    requires requires(Coroutine&& c)
    {
      Ret::get_return_object(static_cast<Coroutine&&>(c));
    }
  static decltype(auto) get_return_object(Coroutine&& coroutine)
    noexcept(noexcept(Ret::get_return_object(
      static_cast<Coroutine&&>(coroutine))))
  {
    return Ret::get_return_object(static_cast<Coroutine&&>(coroutine));
  }
};

template<typename Ret, typename... Args>
  requires requires { typename Ret::promise_type; }
struct coroutine_traits<Ret, Args...>
{
  using promise_type = typename Ret::promise_type;
};
```

The `coroutine_traits` class can be customized by user-code for specific function signatures in the same way that user-code can currently do under the Coroutines TS. This allows non-intrusive adaption of wrapper types to support coroutines. This can be necessary when trying to adapt a return-type from a third-party library where you do not have the ability to modify the source.

The default `coroutine_traits` definition forwards on to the return-type to allow the return type to customize the behavior of the sugar-syntax for coroutines without needing to specialize the `coroutine_traits` type.

The general form of the sugar syntax is the same as for Coroutines TS. If the body contains any of the `co_await`, `co_yield`, `co_return` keywords then the function body is compiled as a coroutine using the sugar syntax.

A function of general form:

```
ReturnType f(ArgTypes... args)
{
  <body-containing-co_xxx-keyword>
}
```

Would be transformed as follows:

```
ReturnType f(ArgTypes... args)
{
  return coroutine_traits<ReturnType, ArgTypes...>::get_return_object(
    [=&] [->] {
      <body-containing-co_xxx-keyword>
    });
}
```

NOTE: Using imaginary `[=&]` syntax here to indicate that all parameters are captured in the coroutine object with the same value-category as the parameter and any moves are perfect-forwarded.

The body of the coroutine is transformed to a call to `coroutine_traits<...>::get_return_object()`, passing the coroutine object constructed from the arguments of the function. Parameters of type lvalue reference or rvalue reference are captured by reference, and by-value parameters are captured by value. The return-value of the call to `get_return_object()` is used as the return-value of the call to the coroutine function.

Note that the default behaviour of the above transformation requires that the full signature is complete and is known in advance and that type returned from `coroutine_traits<...>::get_return_object()` must be convertible to `ReturnType`. This effectively forces the returned wrapper object to type-erase and heap-allocate the coroutine object and/or the coroutine frame – defeating the purpose of exposing the coroutine frame type in the first-place.

What we would really like to do here is specify an alternative 'coroutine_traits' type that contains a `get_return_object()` member function that has a return-type that is deduced from the type of the coroutine object parameter.

Strongly-typed Wrapper Objects

One possible direction to take here would be to allow a function definition to include a trailing coroutine trait-type specifier `[->]` *`CoroutineTraitType`* definition. If so then the `get_return_object()` member function would be looked up within *`CoroutineTraitType`* instead of in `coroutine_traits<Ret, Args...>`.

Deduced return-type from explicit trait type

```
auto foo(int a) [->] static_task<int>
{
  co_return 42;
}

// Transformed to the following:

auto foo(int a)
{
  return static_task<int>::get_return_object(
    [a] [->] { co_return 42; });
}
```

Explicit return-type with explicit trait type

```
task<int> foo(int a) [->] task_builder<int, my_allocator>
{
  co_return 42;
```

```
}

// Transformed to the following:

task<int> foo(int a)
{
  // Return-type of get_return_object() is implicitly converted to
  // return-type of foo().
  return task_builder<int, my_allocator>::get_return_object(
    [a] [->] { co_return 42; });
}
```

Note that the lambda version of this sugar syntax with an explicit trait-type could be considered too syntactically close to the underlying coroutine lambda expression that it sugars over.

For example:

```
void example()
{
  auto range = [](int n) [->] static_generator<int>
  {
    for (int i = 0; i < n; ++i) co_yield i;
  };

  for (int i : range(10))
  {
    std::cout << i << "\n";
  }
}

// Is equivalent to the following

void example_expanded()
{
  auto range = [](int n) {
    return static_generator<int>::get_return_object(
      [n] [->] {
        for (int i = 0; i < n; ++i) co_yield i;
      });
  };
  ...
}
```

Future extension – deducing the trait-type

This approach could possibly be extended in future to allow a deduced trait-type and deduced return type based on inspecting the body of the coroutine once we have vocabulary types in the standard library that we are happy with.

For example, if the return-type is specified as `auto` and no *CoroutineTraitType* is specified and the function body contains at least one of the `co_await`, `co_yield` or `co_return` keywords then:

- If body contains both `co_yield` and `co_await` then use `static_async_generator<T>` trait type where `T` is deduced from argument type to `co_yield`
- If body contains `co_yield` and does not contain `co_await` then use `static_generator<T>` trait type where `T` is deduced from argument type to `co_yield`
- Otherwise use the `static_task<T>` trait type where `T` is deduced from the argument to `co_return`.

Customisation Point Summary

The set of customization points required by this design has been significantly reduced and simplified from the design of the Coroutines TS.

Promise Type	
`final_suspend()`	[Required] Allows the promise type to customize what happens when execution hits the closing curly brace of the coroutine body. Returns a `coroutine_handle` that indicates the continuation that execution should be transferred to.
`unhandled_exception()`	[Optional] Allows the promise type to customize what happens when an unhandled exception escapes the coroutine body. If specified then an implicit `try{ } catch(...) { p.unhandled_exception(); }` is placed around the coroutine body.
`return_value()/return_void()`	[Optional] Allows customizing the behavior of the `co_return` statement within the coroutine body. If coroutine does not contain an explicit or implicit `co_return` statement then this customization point need not be defined (this is expected to be rare – most coroutines should define at least one of these).
`await_value()`	[Optional] Allows customizing the behavior of `co_await` expressions within the coroutine body. If this customization point is not defined then `co_await` expressions within the body are ill-formed.
`yield_value()`	[Optional] Allows customizing the behavior of `co_yield` expressions within the coroutine body. If this customization point is not defined then `co_yield` expressions within the body are ill-formed.
Awaitable	

`operator co_await()`	[Optional] Allows an awaitable type to request additional storage from the coroutine-frame for the duration of the `co_await` expression by returning an `Awaiter` object as a prvalue that is placed as temporary in the coroutine frame. This can allow an `Awaitable` type to implement the `co_await` without need for a separate heap allocation.
Awaiter	
`await_ready()`	[Required] Allows awaiter to returned true to skip execution of compiler-generated suspend-logic and continue execution.
`await_suspend()`	[Required] Allows awaiter to execute some logic after the coroutine has been suspended. Typically, this will schedule resumption of the awaiting coroutine when the result is ready. Can return either `void`, `bool` or `coroutine_handle`. Note: Now that we have `noop_coroutine()` we could potentially eliminate `void` and `bool` variants of `await_suspend()`.
`await_resume()`	[Required] Allows awaiter to customize the result of a `co_await` expression.
Trait Types	
`get_return_object()`	Allows a trait-type to customize how to build the return-value of the function from the coroutine lambda object when using the sugar-syntax. *Note that this could be made optional if we decide to add a coroutine lambda syntax that allows the user to directly construct coroutine objects.*
`coroutine_traits<Ret, Args...>`	[Optional] Allows customization of the default trait type to use for the sugar-syntax when no explicit trait-type is specified. The trait type is deduced based on the signature of the function. This also allows the sugar-syntax to add coroutine support for return-types without intrusively modifying them. The default behavior is to forward `get_return_object()` on to `Ret::get_return_object()` if that function is defined.

Some customization points from the Coroutines TS are no longer required under this proposal and have been removed, others have been simplified:

- `coroutine_traits<...>::promise_type`
  This customization point is no longer needed as the library is responsible for explicitly providing the promise type in a call to `coroutine.create_static<Promise>()` or `coroutine.create_dynamic<Promise>()`.
  The use of `coroutine_traits<>` to customize the behavior based on the signature of the function is still present and has been subsumed by the `get_return_object()` customization point.
- `initial_suspend()`
  The coroutine frame is now always constructed in an initially-suspended state. This allows the library code to choose when to start executing the coroutine and thus we don't need a hook to allow the promise to prevent the coroutine from executing immediately upon creation.
- `final_suspend()`
  This has been simplified from returning an Awaitable type to simply returning a `coroutine_handle` that represents the continuation.
  Note that the coroutine now always suspends when exection hits the end of the coroutine body. The coroutine frame is never implicitly destroyed.
- `promise::get_return_object()`
  This has been replaced by the `trait::get_return_object()` customization point.
- `get_return_object_on_allocation_failure()`
  This is no longer needed since the library is now responsible for calling the `coroutine.create_dynamic()` function and so can handle any errors reported by that call. Further, for contexts where coroutines are being used where dynamic allocation is not allowed or is not allowed to fail, programs can now make use of the `coroutine.create_static<Promise>()` API to implement coroutine creation logic that cannot fail.
- `promise_type::operator new/delete()`
  The library can now customize the dynamic allocation of the coroutine frame by passing an allocator object to the `coroutine.create_dynamic<Promise>()` method. Thus this customization point is no longer required.

Comparison to existing proposals

Differences compared to Coroutines TS

This design retains most of the proven design aspects of the Coroutines TS.

The use of a type-erased `coroutine_handle`, the ability to perform symmetric-transfer between coroutines and the semantics of the `co_await` expression defined in terms of `operator co_await()`, `await_ready()`, `await_suspend()` and `await_resume()` are unchanged.

The main difference is in how the compiler generates the code for creating the coroutine frame. Instead of the compiler composing calls to the allocator, promise constructor, `get_return_object()` and `initial_suspend()` methods in a hard-coded algorithm during creation of the coroutine frame, we expose the individual pieces and let the library compose calls to these pieces themselves.

This, paradoxically, actually has the net effect of reducing the amount of code you need to write to define a new coroutine promise type compared with the Coroutines TS.

We now always create the coroutine frame initially suspended. This means that the library can now choose when to start the coroutine by calling `coroutine_handle::resume()` or *`CoroutineFrameObject`*`::resume()`. This eliminates the need for `initial_suspend()` and so this customization point has been removed.

As the `initial_suspend()` point has been removed and we no longer have the need to keep `final_suspend()` defined in terms of `operator co_await()` for symmetry reasons. Instead, we simplify the `final_suspend()` function to now simply be the equivalent of the `await_suspend()` method on the `Awaitable` object returned from `final_suspend()` in the Coroutines TS. It's purpose now is purely to allow the promise to specify what the continuation of the coroutine should be. The `final_suspend()` method must return a `coroutine_handle` that identifies the continuation. The `noop_coroutine()` handle can be returned if the continuation should be the caller of `resume()` instead of another coroutine.

We have also tweaked the `unhandled_exception()` customization point to now be optional. If it is not specified then any unhandled exceptions that escape the body of the coroutine will be propagated out of the call to `resume()` and there will be no implicit try/catch around the body of the coroutine. The coroutine is considered to be suspended at the final suspend point if an exception exits the coroutine body. This eliminates potential overhead of the try/catch for synchronous generator use-cases which should ideally be transparent to exceptions.

The need for the `operator new/delete()` customisation point on the Coroutines TS has been removed now that we are able to pass an allocator object into the coroutine frame creation function. A nice side-effect of this is that it drastically simplifies the implementation of the memory allocation customisation compared with the current Coroutines TS design.

With the Coroutines TS you would have to sniff out the allocator object from the parameter list (eg. by looking for `std::allocator_arg_t`) and then allocate additional memory on top of what the coroutine frame asked for just so that you can store the allocator object in the extra memory so that you can retrieve it later inside `operator delete()`.

Instead, by passing the allocator object into the coroutine frame creation function, the compiler is able to store the allocator object directly within the coroutine frame structure rather than outside of it through a side-channel.

Differences compared to Core Coroutines

- This design is not as a big change from the mature and well-tested design of Coroutines TS compared with Core Coroutines.
  This means that we can take advantage of a lot of the existing compiler implementation experience as well as the library implementation experience to have a higher level of confidence of the feasibility of this design.
- This design supports guaranteed tail-calls even across translation units, ABI boundaries, debug builds. The restrictions placed on use of `tail return` by the current design of Core Coroutines prevent it from supporting this. This eliminates the dependency on a new experimental '`tail return`' feature proposed in P1063 that would be required with a solution based on Core Coroutines. Instead we reuse the same proven mechanism for tail-calls already adopted in the Coroutines TS and implemented in Clang.
- This design still uses `co_await`/`co_yield`/`co_return` keywords from Coroutines TS. Using the `[<-]`, `[->]`, `return` operators from Core Coroutines is a possibility if there is consensus to do so. The naming of these operators is an orthogonal concern.
- The separation of the steps of creating the coroutine lambda object from the lambda expression and the CoroutineFrame avoids the need for lazy parameters to implicitly wrap the coroutine up in a lambda that becomes a factory for the coroutine frame.
- This design also allows deferring the choice of the promise-type to pass to the coroutine frame creation, even when using the sugar syntax. This enables some interesting and important use-cases. The Core Coroutines design does also allow this, by allowing you to just store the lambda that returns the coroutine frame instead of immediately invoking it.

Design Discussion

Customising memory allocation

Is passing an allocator object to `dynamic_create()` the right approach for customizing allocation compared with an `operator new()`-based solution?

How do we allow the compiler to pass alignment requirements to the call to `allocator.allocate()`?

With `operator new()` we could at least call an overload taking a `std::align_val_t`

Exposing captured parameters from the coroutine object

Do we need to expose access to the captured parameters?
If we do expose them, should we return a tuple of them? a struct with named members?

One use-case is getting hold of implicit object parameter (ie. `*this`).
This could enable an actor-model coroutine type that serialized calls to member functions on the actor object.

eg. Inheriting from actor_base and returning an actor_task the coroutine can automatically serialise calls to the method.

```
class my_actor : public actor_base
{
public:
  my_actor();

  actor_task<bool> try_add(std::string s)
  {
    co_return strings.insert(std::move(s)).second;
  }

  actor_task<bool> try_remove(std::string s)
  {
    co_return strings.erase(s) > 0;
  }

  actor_task<bool> contains(std::string s)
  {
    co_return strings.count(s) > 0;
  }

private:
  std::unordered_set<std::string> strings;
};
```

The `actor_task` class would look at the implicit object parameter in the coroutine object to get access to the operation queue data member stored in `actor_base`.

Copyability of coroutine lambda object

Should the coroutine lambda object be copyable if the captured parameters are copyable? This could allow starting a coroutine from the same starting position multiple times.

Initial feeling is that, yes, this should be supported.

Multi-shot coroutines

Should the *`CoroutineObject`*`::create_static/create_dynamic()` overloads for lvalue be allowed to be called multiple types to create multiple coroutine frames referring to the same coroutine lambda parameter capture pack? Or should these be one-shot?

Customising parameter capture semantics

The current Coroutines TS design captures parameters in the coroutine frame with the same value category as the parameter in the signature.

However, sometimes you don't have control over the signature of the function (eg. you are overriding a virtual function on some base class provided by a third-party) but you want the coroutine to have different capture semantics for particular parameters.

With the Coroutines TS you can alter the capture semantics in an inelegant way by implementing the coroutine in terms of a stateless lambda that has parameters with the correct capture semantics. e.g.

```
task<void> foo(io_service& io, const std::string& s)
{
  // Want to capture 's' by value in the coroutine
  // but continue passing 'io' by reference.
  return std::invoke([](io_service& io, std::string s) -> task<void>
  {
    // body goes here.
  }, io, s);
}
```

The Core Coroutines paper proposes the use of a mandatory lambda capture syntax that requires every coroutine function to explicitly specify both the parameter types and their capture semantics. The motivation behind making this mandatory is to force the developer to think about the capture semantics and to document the decision in code for others.

There are currently some limitations with the ability to specify different value-categories for capture types of a parameter pack. e.g if a function wanted to capture prvalues and xvalues by value and lvalues by reference. However, these limitations I expect to eventually be overcome through tweaks/extensions to the lambda capture syntax.

The larger concern is the need to duplicate the parameter names two or three times in the signature-line of the definition, even when the value categories do not need to change, in order to perfectly forward the values.

eg. the following captures are needed to preserve the defaults in Coroutines TS

```
task<std::string> lookup(io_service& io,
                         std::string key,
                         std::chrono::milliseconds timeout,
                         interrupt_token itoken)
  [&io, key=std::move(key), timeout, itoken=std::move(itoken)] [->]
{
  // body goes here.
}
```

Again, perhaps this can be addressed with future extensions to the lambda capture syntax. eg. by adding a default `[=&&]` syntax which becomes the equivalent of `[x=static_cast<decltype(x)&&>(x)]` for each parameter `x` referenced in the body and not explicitly specified.

Another future direction might be to allow optionally specifying the parameter capture syntax only for parameters where you want to change the default capture semantics. eg. to promote a reference parameter to being captured by value.

Recursive coroutine functions that use a statically-typed wrapper

If you write a coroutine function that recursively calls itself and it returns a wrapper type that makes use of `create_static()` then we can end up with a case where the size of the coroutine frame may depend on the size of the coroutine frame - an unresolvable circular dependency.

eg.

```
auto fib(int n) [->] static_task<int>
{
  if (n < 1) co_return 1;
  co_return co_await fib(n - 1) + co_await fib(n - 2);
}

// It could be multi-level too.

auto foo(int n) [->] static_task<int>;

auto bar(int n) [->] static_task<int>
{
  if (n == 0) co_return 0;

  // Is the definition of foo() required here?
  co_return co_await foo(n - 1);
}

auto foo(int n) [->] static_task<int>
{
  co_return 1 + co_await bar(n - 1);
}
```

Such cases will need to be detected by the compiler and an appropriate error message shown.

The user will need to introduce some kind of heap-allocated wrapper to break the cyclic dependency. eg. replacing one of the return-types in the cycle with a type-erased `task<int>`

Do we really need a coroutine lambda expression syntax?

We might be able to get away with avoiding adding the coroutine lambda expression syntax and instead just providing some way to allow the user to customize the specific trait-type to use for the coroutine independently of the signature of the function.

We could then just use a special "identity" trait object to obtain the coroutine object itself.

```
struct coroutine_object
{
  template<typename Coroutine>
  Coroutine get_return_object(Coroutine&& coroutine)
```

```
  {
    return static_cast<Coroutine&&>(coroutine);
  }
};

auto example2() [->] coroutine_object
{
  co_return 123;
}

void example()
{
  auto coroutine = [] [->] coroutine_object { co_return 123; };
  auto frame = coroutine.create_static<my_promise>();

  auto coroutine2 = example2();
  auto frame2 = coroutine.create_static<my_promise>();
}
```

This could eliminate the need for the `[] [->] { … }` syntax that did not have an explicitly specified trait type.

We would need to consider how this would interact with the coroutine parameter capture specification described above.

```
task<void> foo()
{
  int pos = 0;
  auto makeGenerator =
    // Lambda captures     Parameter captures
    // |                        |
    // V                        V
    [&pos](std::string s) [s=std::move(s)] [->] static_generator<char> {
      while (pos < s.size()) co_yield s[pos++];
    };

  auto g = makeGenerator("hello there!");
}
```

Risks of making unhandled_exception() optional

By making the `unhandled_exception()` method optional it becomes a potential danger that users authoring coroutine promise types fail to define it and thus allow exceptions to propagate out of the coroutine frame. Making it mandatory, however, seems like it violates the "you don't pay for what you don't use" mantra of the C++ community.

Impacts of changes to existing code

The proposed design should allow the majority of existing coroutine functions authored under the Coroutines TS to remain unchanged. Existing Awaitable types that are authored in terms of `coroutine_handle` should also be able to remain unchanged.

This design still represents a breaking change from the Coroutines TS, however. Code that defines coroutine promise_types that are implemented under the Coroutines TS will need to make some changes to their definitions were the proposed design to be adopted. However, a codebase typically has only a small number of coroutine types and so it should be a comparatively small amount of work to port this code to implement the new promise interface.

Some example implementations of coroutine types have been provided in the appendix to this paper to provide some guidance for authors of coroutine types on how to make use of these features.

Future investigation for eliminating the need to choose between `create_static()` and `create_dynamic()` using "deferred layout types"

The rationale behind separating the creation of the coroutine frame into two methods, `create_static()` and `create_dynamic()` is based on the assumption that the coroutine frame type exposed in the C++ type system needs to be able to report a value for `sizeof(FrameType)` in the same way that regular types do. i.e. that it is a constexpr value that is available to the frontend. This then implies that the frontend needs to be able to define the layout of the coroutine frame type before optimisations have executed and thus cannot adjust the layout of the frame to either eliminate storage of state that is unnecessary once inlining and dead-code elimination has been performed or to inline allocation of nested coroutine frames and other heap-allocated objects.

Thus the user is forced to make a tradeoff between creating a coroutine frame that allows inlining of nested allocations and optimised coroutine frame size but can potentially heap allocate, or choosing a statically-known coroutine frame size that is guaranteed not to allocate but that is then not able to inline allocations of nested calls to type-erased coroutines that it calls.

> *Coroutine uncertainty principle: you can either have optimal coroutine frames or you can know the size of the coroutine frame.*
> *- Gor Nishanov*

However, if we relax this assumption and permit the idea of introducing types into the type system where the layout is not known by the frontend and calculation of the layout is deferred until after the middle-end of the compiler has run some optimisation passes, then we can potentially provide a solution that allows both statically typed coroutine frames, with guaranteed stack-allocation AND optimally sized coroutine frames.

Once there is no distinction between the layouts of type-erased and statically-typed coroutine frames then there is no need to force the user to decide between statically allocating and dynamically allocating a coroutine frame unless they need to type-erase the coroutine.

Coroutines could be by-default written as statically-typed coroutines and then only explicitly opted-in to type-erasure when required.

So let's explore the potential definition of a *deferred-layout type (DLT)* in the type system.

A deferred layout type (DLT) is any type that:

- Is a coroutine frame type
- Contains as a non-static data member a DLT
- Has as a base-class a DLT
- Is an array of DTL

The properties of a deferred-layout type, `T`, are:

- `sizeof(T)` is a constant but is not constexpr
- `alignof(T)` is a constant but is not constexpr
- `offsetof(T, member)` is a constant but is not constexpr
- Taking address of a member is not constexpr (the offset of a member from the start of the type is not known at constexpr evaluation time)

Note that DLTs are different from VLAs (variable-length arrays) in that the size of a DLT is a constant by the time the code is lowered into machine code, whereas VLAs are allowed to have a size that is determined dynamically at runtime. Also, VLAs are permitted only in the tail-position of a class when used as a data-member, whereas DLTs would be allowed to be placed anywhere within a class, including multiple distinct DLT data-members.

There would probably need to be some kind of compile-time trait that we could use to ask whether a given type was a deferred-layout type.
e.g. `std::is_deferred_layout_type_v<T>`

If we had a deferred layout type available in the language then we could potentially define the coroutine object interface as follows:

```
template<typename... Captures>
struct Coroutine
{
  std::tuple<Captures&...> get_parameters() &;
  std::tuple<Captures&&...> get_parameters() &&;

  template<bool LvalueCaptures, typename Promise>
  struct coroutine_frame_type
  {
    using coroutine_type =
      std::conditional_t<LvalueCaptures, Coroutine&, Coroutine>;

    template<typename... Args>
      requires Constructible<Promise, Args...> &&
               MoveConstructible<coroutine_type>
    coroutine_frame_type(coroutine_type&& coroutine, Args...&& args)
      : promise_(std::forward<Args&&>(args)...)
```

```
      , coroutine_(std::forward<coroutine_type>(coroutine))
      , /* other compiler-generated initialisation */
    {}

    ~coroutine_frame_type();

    coroutine_frame_type(coroutine_frame_type&&) = delete;
    coroutine_frame_type(const coroutine_frame_type&) = delete;
    coroutine_frame_type& operator=(const coroutine_frame_type&) = delete;
    coroutine_frame_type& operator=(coroutine_frame_type&&) = delete;

    bool done() const;

    void resume();

    coroutine_handle<Promise> get_handle() noexcept;

    std::tuple<Captures&...> get_parameters();

  private:

    Promise promise_;
    coroutine_type coroutine_;

    // compiler defined layout (deferred-layout type)
  };
};
```

Then we could define `create_static()` as library helper function:

```
template<typename Promise, typename Coroutine, typename... Args>
decltype(auto) create_static(Coroutine&& coroutine, Args&&... args)
{
  using coroutine_frame_type =
    typename std::remove_reference_t<Coroutine>::
    template coroutine_frame_type<
      std::is_lvalue_reference<Coroutine>,
      Promise>;
  return coroutine_frame_type{
    std::forward<Coroutine>(coroutine),
    std::forward<Args>(args)...};
}
```

It may still be necessary to define a compiler-generated create_dynamic() function that allowed the caller to heap-allocate the coroutine frame in a way that would allow the compiler to still elide the allocation and that would type-erase the allocator used to allocate the frame. However, the layout of the coroutine frame for create_dynamic() should not be any different when created with create_dynamic() vs the frame created with create_static().

Other issues that need to be explored to determine the viability of this approach:

- How much of existing standard library implementations would break if they could not assume `sizeof(T)` was always constexpr?

- Are there any potential ODR-violations relating to having the same coroutine frame type be referenced from multiple translation-units?
- How do we guarantee that a given coroutine frame type always has the same layout across all translation-units? eg. if a class defined in a header has an inline method that returns a `static_task<T, Coroutine>` then how does the linker ensure that different translation units that include that header all see the same layout for `Coroutine`?

A path for coroutines in C++20

Given that this proposal makes some non-trivial design changes to the Coroutines TS it seems unlikely it would be feasible to adopt it in entirety, implement, test, finalize and ship it for C++20.

So the challenge then becomes defining a subset of this design that allows us to ship an initial functional implementation in C++20 but still allow for the full set of functionality in a future standard without breaking backwards compatibility.

Our recommended path is to define for C++20 only the type-erased sugar syntax version of coroutines with syntax and semantics much like the Coroutines TS and then use the coroutine_traits customization point to trigger different default behaviours.

For C++20 we define coroutines to work as follows:

We allow the user to customize the `coroutine_traits` template to define a custom `promise_type`. Then we transform the coroutine function similarly to how it currently does under the Coroutines TS – using a hard-coded algorithm that allocates the coroutine frame, constructs the promise, etc. all in terms of methods on the `promise_type`. We might need some tweaks to this algorithm to make it future compatible, but could be done.

Then in C++Next we add the following logic:

If the compiler encounters a coroutine that uses the new syntax that explicitly specifies the *`CoroutineTraitType`* to use then its body is translated into a call to *`CoroutineTraitType`*`::get_return_object(coroutineObject)`

Otherwise, if the compiler encounters a coroutine that uses the sugar syntax and doesn't explicitly define a *`CoroutineTraitType`* using C++Next-specific syntax then the compiler consults the `coroutine_traits` customization point.

If `coroutine_traits<Ret, Args...>::promise_type` is defined then the compiler dispatches to some internal `get_return_object()` implementation that implements the

hard-coded algorithm from C++20 in terms of the new low-level coroutine APIs. e.g. something like:

```
template<typename Promise, typename Coroutine>
auto __cpp20_get_return_object(Coroutine&& coroutine)
{
  return __cpp20_get_return_object_impl<Promise>(
    std::forward<Coroutine>(coroutine),
    std::move(coroutine.get_parameters()));
}

template<typename Promise, typename Coroutine, typename... Args>
auto __cpp20_get_return_object_impl(
  Coroutine&& coroutine, std::tuple<Args...>&& parameters)
{
  auto allocator = std::apply([](Args&&... args) {
    return Promise::get_allocator(static_cast<Args&&>(args)...);
  }, std::move(parameters));

  // TODO: Allow parameter peeking with promise.
  // Problem with getting post-moved parameters into promise constructor.

  auto handle =
    static_cast<Coroutine&&>(coroutine)
    .create_dynamic<Promise>(std::move(allocator));

  return promise.get_return_object(handle);
}
```

Otherwise, if `coroutine_traits<Ret, Args...>::get_return_object()` is defined then the compiler simply translates the coroutine function into a call to that `get_return_object()`, passing the coroutine object.

Then we modify the `coroutine_traits<Ret, Args...>` primary template definition such that:

- If `Ret::promise_type` typedef is defined then `coroutine_traits::promise_type` is a nested type alias defined to be equal to `Ret::promise_type`.
- Otherwise, if `Ret::get_return_object()` is defined then the `coroutine_traits` type will contain a single static `get_return_object()` function that calls `Ret::get_return_object()`.
- Otherwise, `coroutine_traits` contains no nested members.

This should now allow C++20-style coroutines and C++Next-style coroutines to exist side-by-side.

Further consideration is required:

- as to the potential ABI-compatibility issues relating to such a transition.
- as to whether the added complexity of having two different flavours of promise type will be confusing for developers when it is introduced in C++Next.

Conclusion

The design presented in this paper proposes an evolution of the Coroutines TS design that allows programs to create coroutines that are guaranteed not to be heap-allocated – one of the key complaints of the current Coroutines TS design.

The scope of this change is likely too large to be able to ship in C++20 but there is a potential path to shipping part of this proposal in C++20 with support for type-erased coroutine frames (which is what we have in Coroutines TS already), while still allowing an evolution to future support for guaranteed stack-allocated coroutines in a future ship vehicle.

Appendix - Example Coroutine Types

Below are listed several example implementations of coroutine types to demonstrate how we can leverage the customization points to define different types of coroutine semantics – both type-erased and guaranteed no-allocation varieties.

We can even allow implicit conversion from non-type-erased to type-erased versions so that we can allow coroutines functions to be defined using non-type-erased implementation but still be able to pass the result into a function that takes a type-erased version. See static_task<T> and task<T>.

A `std::optional<T>` coroutine

```
template<typename T, typename... Args>
struct coroutine_traits<std::optional<T>, Args...>
{
private:
  struct promise
  {
    std::optional<T>& result;

    auto final_suspend() noexcept { return noop_coroutine(); }

    template<ConvertibleTo<T> U>
    void return_value(U&& value)
    {
      result.emplace(static_cast<U&&>(value));
    }

    template<typename Optional>
    struct awaiter
    {
      Optional&& opt;

      bool await_ready() noexcept { return opt.has_value(); }
      void await_suspend(coroutine_handle<promise>) {}
      decltype(auto) await_resume() noexcept {
        return static_cast<Optional&&>(opt).value();
      }
    };
```

```
    template<typename U>
    auto await_value(std::optional<U>& opt) noexcept
    {
      return awaiter<std::optional<U>&>{ opt };
    }

    template<typename U>
    auto await_value(std::optional<U>&& opt) noexcept
    {
      return awaiter<std::optional<U>>{ std::move(opt) };
    }

    template<typename U>
    auto await_value(const std::optional<U>& opt) noexcept
    {
      return awaiter<const std::optional<U>&>{ opt };
    }

    template<typename U>
    auto await_value(const std::optional<U>&& opt) noexcept
    {
      return awaiter<const std::optional<U>>{ std::move(opt) };
    }
  };

public:

  template<typename Coroutine>
  static std::optional<T> get_return_object(Coroutine&& coroutine)
  {
    std::optional<T> result;

    // Guaranteed no heap allocations.
    coroutine.create_static<promise>(result).resume();

    // Named-return-value optimization compatible.
    return result;
  }
};
```

An eager oneway_task

```
class [[maybe_unused]] oneway_task
{
  struct promise
  {
    auto final_suspend(coroutine_handle<promise> h) noexcept
    {
      h.destroy();
      return noop_coroutine();
    }

    void return_void() noexcept {}

    [[noreturn]] void unhandled_exception() noexcept { std::terminate(); }
```

```
    template<typename T>
    T&& await_value(T&& value) { return static_cast<T&&>(value); }
  };

public:
  template<typename Coroutine>
  static oneway_task get_return_object(Coroutine&& coroutine)
  {
    static_cast<Coroutine&&>(coroutine)
      .create_dynamic<promise>(std::allocator<char>{}).resume();
    return oneway_task{};
  }
};
```

A non-type-erased coroutine - `static_task<T>`

```
template<typename T, typename Coroutine = void>
class static_task
{
  Coroutine coroutine;

  struct promise
  {
    coroutine_handle<void> continuation;
    std::optional<T> value;
    std::exception_ptr ex;

    void unhandled_exception() noexcept { ex = std::current_exception(); }

    template<ConvertibleTo<T> U>
    void return_value(U&& x) { value.emplace(static_cast<U&&>(x)); }

    auto final_suspend(coroutine_handle<promise>) noexcept
    {
      return continuation;
    }

    template<typename T>
    T&& await_value(T&& value) { return std::forward<T>(value); }
  };

  using coroutine_frame =
    decltype(std::declval<Coroutine>().create_static<promise>());

public:
  explicit static_task(Coroutine&& coroutine)
  : coroutine(static_cast<Coroutine&&>(coroutine))
  {}

  static_task(static_task&& other) = default;

  auto operator co_await() &&
  {
```

```
    struct awaiter
    {
      coroutine_frame frame;

      awaiter(Coroutine&& coroutine)
      : frame(static_cast<Coroutine&&>(coroutine).create_static<promise>())
      {}

      bool await_ready() noexcept { return false; }

      auto await_suspend(coroutine_handle<> continuation) noexcept
      {
        frame.promise().continuation = continuation;
        return frame.get_handle();
      }

      T await_resume() {
        if (frame.promise().ex) {
          std::rethrow_exception(std::move(frame.promise().ex));
        }
        return std::move(frame.promise().value).value();
      }
    };

    // Relying on guaranteed copy-elision here so that the awaiter
    // object is constructed in-place in the final location.
    return awaiter{ std::move(coroutine) };
  }
};

template<typename T>
class static_task<T, void>
{
public:
  template<typename Coroutine>
  static auto get_return_object(Coroutine&& coroutine)
  {
    return static_task<T, Coroutine>{ std::move(coroutine) };
  }
};
```

A type-erased coroutine - `task<T>`

```
template<typename T>
class task
{
  class promise
  {
    coroutine_handle<> continuation;
    std::optional<T> value;
    std::exception_ptr ex;

    auto final_suspend(coroutine_handle<promise>) { return continuation; }
```

```
    void unhandled_exception() noexcept { ex = std::current_exception(); }

    void return_value(T v) { value.emplace(std::move(v)); }

    template<typename T>
    T&& await_value(T&& v) { return std::forward<T>(v); }
  };

  coroutine_handle<promise> handle;

public:

  // Enable use of sugar syntax.
  template<typename Coroutine>
  static task<T> get_return_object(Coroutine&& coroutine)
  {
    return task<T>{ static_cast<Coroutine&&>(coroutine) };
  }

  template<typename Coroutine>
  explicit task(Coroutine&& coroutine) noexcept {
  : handle(static_cast<Coroutine&&>(coroutine).create_dynamic<promise>())
  {}

  // Implicit conversion from static_task<T, Coroutine>
  template<typename Coroutine>
  task(static_task<T, Coroutine>&& task)
  : task(static_cast<Coroutine&&>(task.coroutine))
  {}

  task(task&& t) noexcept : handle(std::exchange(t.handle, {})) {}

  ~task() { if (handle) handle.destroy(); }

  auto operator co_await() && noexcept
  {
    struct awaiter {
      coroutine_handle<promise> handle;

      bool await_ready() noexcept { return false; }

      auto await_suspend(coroutine_handle<> continuation) noexcept
      {
        handle.promise().continuation = continuation;
        return handle;
      }

      T await_resume()
      {
        if (handle.promise().ex) {
          std::rethrow_exception(std::move(handle.promise().ex));
        }
        return *std::move(handle.promise().value);
      }
    };

    return awaiter{ handle };
```

```
  }
};
```

A type-erased `generator<T>`

```
struct suspend_always
{
  bool await_ready() { return false; }
  template<typename Handle> void await_suspend(Handle) noexcept {}
  void await_resume() noexcept {}
};

template<typename T>
class generator {
  struct promise {
    std::add_pointer_t<T> value;

    void return_void() noexcept { value = nullptr; }

    auto final_suspend(coroutine_handle<promise>) noexcept {
      return noop_coroutine();
    }

    suspend_always yield_value(T& value) noexcept {
      this->value = std::addressof(value);
      return {};
    }

    suspend_always yield_value(T&& value) noexcept {
      this->value = std::addressof(value):
      return {};
    }
  };

  coroutine_handle<promise> handle;

public:
  template<typename Coroutine>
  static generator<T> get_return_object(Coroutine&& coroutine)
  {
    return generator<T>{ static_cast<Coroutine&&>(coroutine) };
  }

  template<typename Coroutine>
  generator(Coroutine&& coroutine)
  : handle(coroutine.create_dynamic<promise>())
  {}

  struct sentinel {};

  struct iterator
  {
    coroutine_handle<promise> handle;
    iterator& operator++() { handle.resume(); return *this; }
    T& operator*() noexcept { return *handle.promise().value; }
```

```
    bool operator==(sentinel) noexcept { return handle.done(); }
    bool operator!=(sentinel end) noexcept { return !operator==(end); }
  };

  iterator begin() { handle.resume(); return iterator{ handle }; }
  sentinel end() { return {}; }

};
```

A statically typed generator – static_generator<T>

```
template<typename T, typename Coroutine = void>
struct static_generator
{
  struct promise {
    std::add_pointer_t<T> value;

    void return_void() noexcept { value = nullptr; }

    auto final_suspend(coroutine_handle<promise>) noexcept {
      return noop_coroutine();
    }

    suspend_always yield_value(T& value) noexcept {
      this->value = std::addressof(value);
      return {};
    }

    suspend_always yield_value(T&& value) noexcept
      requires !std::is_lvalue_reference_v<T>
    {
      this->value = std::addressof(value):
      return {};
    }
  };

  using coroutine_frame =
    decltype(std::declval<Coroutine&>().create_static<promise>());

  // Construction of frame is deferred to allow coroutine to be moved
  // up until begin() is called.
  Coroutine coroutine;
  union {
    coroutine_frame frame;
  };
  bool frameCreated = false;

public:

  template<typename Coroutine>
  static auto get_return_object(Coroutine&& coroutine)
  {
    return static_generator<T, Coroutine>{
      std::forward<Coroutine>(coroutine) };
  }
```

```
  template<typename Coroutine>
  explicit static_generator(Coroutine&& coroutine)
  : coroutine(static_cast<Coroutine&&>(coroutine))
  {}

  static_generator(static_generator&& other)
  : coroutine(std::move(other.coroutine))
  {
    assert(!other.frameCreated);
  }

  ~static_generator()
  {
    if (frameCreated) frame.~coroutine_frame();
  }

  struct sentinel {};

  struct iterator
  {
    coroutine_frame& frame;
    iterator& operator++() { frame.resume(); return *this; }
    T& operator*() noexcept { return *frame.promise().value; }
    bool operator==(sentinel) const noexcept { return frame.done(); }
    bool operator!=(sentinel) const noexcept { return !frame.done(); }
  };

  iterator begin()
  {
    assert(!frameCreated);
    ::new (static_cast<void*>(&frame)) coroutine_frame{
      coroutine.create_static<promise>() };
    frameCreated = true;

    frame.resume();
    return iterator{ frame };
  }

  sentinel end() { return {}; }
};
```

A type-erased `async_generator<T>`

```
template<typename T>
class async_generator
{
  class promise
  {
    std::add_pointer_t<T> value;
    std::exception_ptr error;
    coroutine_handle<> continuation;

    struct awaiter
```

```
    {
      bool await_ready() noexcept { return false; }
      auto await_suspend(coroutine_handle<promise> h) noexcept
      {
        return h.continuation;
      }
      void await_resume() noexcept {}
    };

    auto final_suspend(coroutine_handle<promise>) noexcept
    {
      return continuation;
    }

    template<typename U>
    U&& await_value(U&& value) { return static_cast<U&&>(value); }

    awaiter yield_value(T& value) noexcept
    {
      this->value = std::addressof(value);
      return {};
    }

    awaiter yield_value(T&& value) noexcept
      requires !std::is_lvalue_reference_v<T>
    {
      this->value = std::addressof(value);
      return {};
    }

    void return_void() noexcept {}

    void unhandled_exception() noexcept
    {
      error = std::current_exception();
    }
  };

  coroutine_handle<promise> handle;

public:
  template<typename Coroutine>
  explicit async_generator(Coroutine&& coroutine)
  : handle(static_cast<Coroutine&&>(coroutine).create_dynamic<promise>())
  {}

  async_generator(async_generator&& other) noexcept
  : handle(std::exchange(other.handle, {}))
  {}

  ~async_generator()
  {
    if (handle) handle.destroy();
  }

  struct sentinel {};
```

```
  struct async_iterator
  {
    coroutine_handle<promise> handle;

    auto operator++() noexcept
    {
      struct awaiter
      {
        async_iterator& it;

        bool await_ready() { return false; }

        auto await_suspend(coroutine_handle<> continuation)
        {
          it.handle.promise().continuation = continuation;
          return it.handle;
        }

        async_iterator& await_resume()
        {
          auto& error = it.handle.promise().error
          if (error) std::rethrow_exception(error);
          return it;
        }
      };

      return awaiter{ *this };
    }

    T& operator*() noexcept { return *handle.promise().value; }

    bool operator==(sentinel) const { return handle.done(); }
    bool operator!=(sentinel) const { return !handle.done(); }
  };

  auto begin() noexcept
  {
    struct awaiter
    {
      coroutine_handle<promise> handle;

      bool await_ready() noexcept { return false; }

      auto await_suspend(coroutine_handle<> continuation) noexcept
      {
        handle.promise().continuation = continuation;
        return handle;
      }

      async_iterator await_resume()
      {
        auto& error = handle.promise().error;
        if (error) std::rethrow_exception(error);
        return async_iterator{ handle };
      }
    };
```

```
    return awaiter{ handle };
  }

  sentinel end() { return {}; }

};
```

A static_task<T> type that allows the nested coroutine to inline the resumption of the caller

This implementation relies on a hypothetical future extension to the coroutines proposal that allows a coroutine to pass a strongly-typed coroutine_handle to the await_suspend() method that carries with it static type information that identifies the specific coroutine and suspend-point of the awaiting coroutine's continuation.

eg. imagine that instead of passing a coroutine_handle<Promise> it passed a type coroutine_handle<Promise, SuspendPoint> such that calling .resume() on that coroutine handle would be able to be translated to a direct jump into that position without needing to lookup state in the coroutine frame to determine the current suspend-point.

```
template<typename T, typename Coroutine = void>
class static_task
{
  Coroutine coroutine;

  struct promise_base
  {
    std::optional<T> value;
    std::exception_ptr ex;

    template<typename T>
    T&& await_value(T&& value) { return std::forward<T>(value); }

    void unhandled_exception() noexcept { ex = std::current_exception(); }

    template<ConvertibleTo<T> U>
    void return_value(U&& x) { value.emplace(static_cast<U&&>(x)); }
  };

  template<typename CoroutineHandle>
  struct promise : promise_base
  {
    const CoroutineHandle continuation;

    explicit promise(CoroutineHandle continuation) noexcept
    : continuation(continuation)
    {}

    auto final_suspend(coroutine_handle<promise>) noexcept
    {
      return continuation;
    }
```

```
  };

  // Calculate the size of the coroutine frame required based on a
  // type-erased coroutine handle continuation type. The compiler
  // should end up producing the same coroutine frame layout regardless
  // of the type of the continuation.
  // We verify this with a static_assert() this inside the await_suspend()
  // method.
  using coroutine_frame_prototype =
    decltype(std::declval<Coroutine&>()
             .create_static<promise<coroutine_handle<void>>>());

public:
  explicit static_task(Coroutine&& coroutine)
  : coroutine(static_cast<Coroutine&&>(coroutine))
  {}

  static_task(static_task&& other) = default;

  auto operator co_await() &&
  {
    struct awaiter
    {
      using deleter_func = void(void*);

      Coroutine& coroutine;
      promise_base* promise = nullptr;

      alignas(coroutine_frame_prototype)
      char frameStorage[sizeof(coroutine_frame_prototype)];

      deleter_func* deleter = nullptr;

      awaiter(Coroutine& coroutine)
      : coroutine(coroutine)
      {}

      ~awaiter()
      {
        // Unfortunately we have to type-erase the frame destruction here.
        // To solve this we'd need access to the suspend-point handle as
        // a parameter to operator co_await().
        if (deleter) { deleter(&frameStorage); }
      }

      bool await_ready() noexcept { return false; }

      template<typename CoroutineHandle>
      auto await_suspend(CoroutineHandle continuation) noexcept
      {
        using coroutine_frame =
          decltype(coroutine.create_static<promise<CoroutineHandle>>());

        // Compile-time sanity checks.
        static_assert(sizeof(coroutine_frame) ==
                      sizeof(coroutine_frame_prototype));
        static_assert(alignof(coroutine_frame) ==
```

```
                        alignof(coroutine_frame_prototype));

        auto* frame =
          ::new (static_cast<void*>(&frameStorage)) coroutine_frame{
            coroutine.create_static<promise<CoroutineHandle>>(continuation)
          };

        deleter = [](void* ptr)
        {
          static_cast<coroutine_frame*>(ptr)->~coroutine_frame();
        };
        promise = &frame->promise();

        return frame->get_handle();
      }

      T await_resume() {
        if (promise->exception) {
          std::rethrow_exception(std::move(promise->exception));
        }
        return std::move(promise->value).value();
      }
    };

    return awaiter{ coroutine };
  }
};

template<typename T>
class static_task<T, void>
{
public:
  template<typename Coroutine>
  static auto get_return_object(Coroutine&& coroutine)
  {
    return static_task<T, Coroutine>{ std::move(coroutine) };
  }
};
```