Document No.	P1288R0
Date	2018-10-07
Reply To	Lewis Baker <lbaker@fb.com>
Audience	SG1, LEWG

Coroutine concepts and metafunctions

Abstract

The Coroutines TS introduces the ability to `co_await` a value from within a coroutine.

When building generic functions and types that interact with coroutines you will often want to constrain a generic function to only accept parameters to which you can apply the `co_await` operator. You may also want to query the result-type of a `co_await` expression so that you can use it to construct the correct return-type of a coroutine.

This can be quite common when building adapters for awaitable types such as those that save/restore context across coroutine suspend-points, perform logging or that schedule execution of an awaitable onto a different execution context. Other usage examples include sync_wait() from P1171R0 and when_all()/when_all_ready().

It would be much simpler to write correct generic coroutine code if this could be written in terms of concepts and type-traits that were available in the standard library:

```
template<Executor E, Awaitable A>
auto schedule_on(E executor, A awaitable) -> task<await_result_t<A>>
{
  co_await executor.schedule();
  co_return co_await std::move(awaitable);
}
```

Currently, it is not possible to write this sort of constrained generic code without writing a number of subtly complicated trait types that emulate the logic the compiler applies when compiling a `co_await` expression.

This paper proposes to add:

- New concept definitions to the standard library that allow functions to be constrained to accepting only awaitable types: `Awaiter`, `AwaiterOf<R>`, `Awaitable`, `AwaitableOf<R>`
- New trait types to the standard library that allow generic code to query the result-type of a `co_await` expression and to query the type of the intermediate awaiter object returned from calling `operator co_await()`: `await_result_t<T>`, `awaiter_type_t<T>`
- A `get_awaiter()` helper function that lets library code simulate the same rules as the compiler for obtaining an 'awaiter' object from an 'awaitable' object. This helper is useful in building awaitable adapters.

Background

One of the limitations of `operator co_await` that is specified in N4760 8.3.8(2) is that it may only appear in a potentially evaluated context. This means that we cannot simply use `decltype(co_await foo)` to determine the result-type of a `co_await` expression.

The reason this restriction exists is because the result of a `co_await` expression is context-dependent. The validity of a `co_await` expression and its resulting type depends on the `promise_type` of the specific coroutine in which it is evaluated and whether there is an `await_transform()` member on the `promise_type`.

A type that is awaitable in one coroutine context may not be awaitable in another coroutine context if those coroutines have different promise types. Further, the result-type of a `co_await` expression in one coroutine-context may be different from the result-type of a `co_await` of the same expression in a different coroutine-context due to the promise-types having different `await_transform()` implementations, even though the operand type is the same.

This means that when asking a question about whether or not a given type is awaitable or asking what the result-type of a `co_await` expression will be, you need to qualify the question with the context in which the `co_await` expression will be evaluated. You can either ask whether a type is awaitable within a specific context (eg. within a coroutine that has a specific/known promise_type) or you can ask whether a type is awaitable within a general context where you assume certain properties about the `promise_type` (eg. assuming the `promise_type` does not define an `await_transform()` member).

Further, when determining the result of a `co_await` expression, we need to handle three different cases:

- Where the awaitable object has a member `operator co_await()`.
- Where the awaitable object has a non-member `operator co_await()`.
- Where the awaitable object does not have an `operator co_await()`.
  In this case the `await_ready()`, `await_suspend()` and `await_resume()` methods are found directly on the awaitable object.

The semantics of a ‘`co_await <expr>`’ expression can be (roughly) summarised by the following pseudo-code:

```
{
  auto&& awaitedValue = <expr>;
  auto&& awaitable = get_awaitable(
    promise, static_cast<decltype(awaitedValue)>(awaitedValue));
  auto&& awaiter = get_awaiter(
    static_cast<decltype(awaitable)>(awaitable));
  if (!awaiter.await_ready())
  {
    <suspend-coroutine>

    using handle_type = std::experimental::coroutine_handle<promise_type>;
    handle_type coro = handle_type::from_promise(promise);

    using await_suspend_result = decltype(awaiter.await_suspend(coro));
    if constexpr (std::is_void_v<await_suspend_result>)
    {
      awaiter.await_suspend(coro);
      <return-to-caller-or-resumer>
    }
    else if constexpr (std::is_same_v<await_suspend_result, bool>)
    {
      if (awaiter.await_suspend(coro))
      {
        <return-to-caller-or-resumer>
      }
    }
    else
    {
      static_assert(__is_coroutine_handle_v<await_suspend_result>);
      awaiter.await_suspend(coro).resume(); // tail-call
      <return-to-caller-or-resumer>
    }

    <resume-point>
  }
  awaiter.await_resume();
}
```

Assuming that following helper functions are defined:

```
template<typename Promise, typename Value>
decltype(auto) get_awaitable(Promise& promise, Value&& value)
```

```
{
  if constexpr (__has_await_transform_member_v<Promise>)
  {
    return promise.await_transform(static_cast<Value&&>(value));
  }
  else
  {
    return static_cast<Value&&>(value);
  }
}

template<typename Awaitable>
decltype(auto) get_awaiter(Awaitable&& awaitable)
{
  if constexpr (__has_member_operator_co_await_v<Awaitable>)
  {
    return static_cast<Awaitable&&>(awaitable).operator co_await();
  }
  else if constexpr (__has_free_operator_co_await_v<Awaitable>)
  {
    return operator co_await(static_cast<Awaitable&&>(awaitable));
  }
  else
  {
    return static_cast<Awaitable&&>(awaitable);
  }
}
```

Terminology

This section defines some placeholder terminology that we can use when talking about different parts of a `co_await` expression. For a more in-depth discussion of the semantics of `operator co_await` see the blog-post Understanding operator co_await[1].

Natural coroutine context - A coroutine context where the `promise_type` of the coroutine does not have an `await_transform()` method and so `co_await` expressions within that context have their natural meaning/semantics.

[1] https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await

Modified coroutine context - A coroutine context where the behaviour and/or result-type of a `co_await` expression may be affected by the presence of an `await_transform()` method on the coroutine's `promise_type`.

Awaited value - The result of evaluating the sub-expression in the operand position of the `co_await` expression.

Awaitable - Something that you can apply the 'co_await' operator to. If the promise type defines an await_transform() member then the awaitable is obtained by calling `promise.await_transform(value)`, passing the awaited value. Otherwise, if the promise type does not define an `await_transform()` member then the awaitable is the awaited value itself.

Awaiter - An awaiter object is an object that implements the await_ready(), await_suspend() and await_resume() methods. If the awaitable object implements either a member or non-member `operator co_await()` then the awaiter object is the result of calling `operator co_await()`. Otherwise, the awaiter object is the same as the Awaitable object.
Note that in P1056R0, the proposed name for this concept was SimpleAwaitable.

The Awaitable/Awaiter terminology has precedent[2] in the .NET Framework.
The C# await keyword maps to a call to awaitable.GetAwaiter() to obtain an Awaiter object. The Awaiter object has an `IsCompleted` property, `OnCompleted()` method that takes a continuation, and a `GetResult()` method for obtaining the final result. These three components of the interface of an Awaiter object in .NET have direct mappings, respectively, to the `await_ready()`, `await_suspend()` and `await_resume()` methods described in N4760[3].

[2] https://weblogs.asp.net/dixin/understanding-c-sharp-async-await-2-awaitable-awaiter-pattern
[3] https://wg21.link/N4760

API Synopsis

Concepts

```
// <experimental/coroutine> header
namespace std::experimental
{
  // A type, T, matches this concept if it implements valid overloads
  // for await_ready(), await_suspend() and await_resume() in _any_
  // natural coroutine context (ie. await_suspend() accepts an argument
  // of type coroutine_handle<void>)
  template<typename T>
  concept Awaiter;

  // A type, T satisfies AwaiterOf<T, R> if a value of type T& implements
  // the await_ready(), await_suspend() and await_resume() methods and the
  // result of await_resume() is convertible to type R.
  template<typename T, typename R>
  concept AwaiterOf;

  // A type, T, is Awaitable if, given an expression, E, of type, T, then
  // the expression ‘co_await E’ is a valid expression within _any_
natural
  // coroutine context. Such a type must either have a member or
non-member
  // operator co_await() or must be an Awaiter.
  template<typename T>
  concept Awaitable;

  // A type T, is AwaitableOf<T, R> if it is Awaitable and the result of
  // the co_await expression is convertible to R.
  template<typename T, typename R>
  concept AwaitableOf;
}
```

Trait Types

```
// <experimental/coroutine> header
namespace std::experimental
{
  // Query the result-type of a co_await expression with an operand of
  // type T.
  template<typename T>
  struct await_result
  {
    using type = ...;
  };

  template<typename T>
  using await_result_t = typename await_result<T>::type;

  // Query the type returned from calling member or non-member
  // operator co_await() on a value of type, T, if operator co_await()
  // is defined, otherwise yields the type T&&.
  template<typename T>
  struct awaiter_type
  {
    using type = ...;
  };

  template<typename T>
  using awaiter_type_t = typename awaiter_type<T>::type;
}
```

Functions

```
// <experimental/coroutine> header
namespace std::experimental
{
  // Returns the result of applying operator co_await() to the function's
  // argument, if the operator is defined, otherwise returns a reference
  // to the input argument.
  template<Awaitable T>
  auto get_awaiter(T&& awaitable) -> awaiter_type_t<T>;
}
```

Design Discussion

`Awaiter` Concept

The `Awaiter` concept is used to refer to types that implement the trio of methods; `await_ready()`, `await_suspend()` and `await_resume()` that are called by a coroutine when evaluating a `co_await` expression.

The concept requires that these methods be callable on an lvalue-reference to a value of the type. This mirrors the requirement from the wording within N4760 8.3.8(3) which states that these methods are called on an lvalue-reference to the awaiter object.

This concept is restricted to matching types that can be awaited from an arbitrary *natural coroutine context*. This is implemented by checking that the `await_suspend()` method is callable with a single parameter of type `coroutine_handle<void>` type. If the `await_suspend()` method is able to accept a `coroutine_handle<void>` type then it will also be able to accept a `coroutine_handle<P>` type for an arbitrary promise type, `P`, as `coroutine_handle<P>` inherits from `coroutine_handle<void>` and thus is implicitly convertible.

The rationale here is that for an awaiter object to be able to support being awaited in an arbitrary natural coroutine context it will generally need to type-erase the `coroutine_handle<Promise>` to `coroutine_handle<void>` so that it can store the continuation for an arbitrary coroutine-type. If the `await_suspend()` method overload-set only has overloads that accept specific types of `coroutine_handle<P>` then it is only awaitable within specific contexts and thus we don't consider it to satisfy the `Awaiter` concept.

`Awaitable` Concept

The awaitable concept simply checks whether the type supports applying the `co_await` operator to a value of that type.

If the object has either a member or non-member `operator co_await()` then its return value must satisfy the `Awaiter` concept. Otherwise, the `Awaitable` object must satisfy the `Awaiter` concept itself.

`AwaiterOf<T>` Concept

This concept subsumes `Awaiter` and places the additional constraint that the result of the `co_await` expression (ie. the return-value of `await_resume()`) is convertible to type, `T`.

`AwaitableOf<T>` Concept

This concept subsumes `Awaitable` and places the additional constraint that the result of the `co_await` expression is convertible to type, `T`.

`await_result<T>` Type Trait

This type trait is used to compute the result-type of a co_await expression where the operand has type, T. If type, T, satsifies the Awaitable concept then `await_result<T>` will contain a nested ‘`type`’ typedef that will be equal to the result-type of the `co_await` expression. Otherwise, if `T` does not satisfy the `Awaitable` concept then `await_result<T>::type` will not be defined.

Example:

```
template<Awaitable A>
task<await_result_t<A>> make_task(A awaitable)
{
  co_return co_await static_cast<A&&>(awaitable);
}
```

`awaiter_type<T>` Type Trait

This type trait computes the return-type of the `get_awaiter()` function when passed a value of type, `T`.

This is typically used when building awaitable/awaiter types that adapt other awaitable types.

Example:

```
template<Awaitable A>
struct logging_awaitable
{
  A awaitable;

  auto operator co_await() &&
  {
    struct logging_awaiter
    {
      awaiter_type_t<A> awaiter; // <- Usage example here
      bool suspended = false;

      awaiter(A&& awaitable)
      :
awaiter(std::experimental::get_awaiter(static_cast<A&&>(awaitable))
       {}

       decltype(auto) await_ready() { return awaiter.await_ready(); }

       decltype(auto) await_suspend(std::experimental::coroutine_handle<>
h)
       {
         LOG("Suspended");
         suspended = true;
         return awaiter.await_suspend(h);
       }

       decltype(auto) await_resume()
       {
         if (suspended) { LOG("Resumed"); }
         return awaiter.await_resume();
       }
     };

     return logging_awaiter{ static_cast<A&&>(awaitable) };
   }
 };
```

`get_awaiter()` Function

The get_awaiter() function is useful for building new operators, when passed an object that has an associated operator co_await() defined for that type then it returns the result of calling operator co_await(), otherwise it returns the argument unmodified.

Note that this function does not constrain the argument to have to satisfy the Awaitable concept as we also want this function to be usable to obtain the awaiter object for types that are only awaitable within specific coroutine contexts.

Asking whether a type is awaitable within a specific context

The Background section of this proposal mentions that there are two kinds of questions we could be asking about whether a type is awaitable. The above concepts and trait types allow you to answer the first question; “is this type awaitable within an arbitrary natural coroutine context?”.

The other question not covered by this proposal is asking “is this type awaitable within this specific coroutine context?”. It was an intentional decision to leave this out for now for a couple of reasons.

Firstly, the use case for asking this question is not as clear as for a general coroutine context. Typically, if you know that you will be awaiting a type within a specific type of coroutine then you often also have knowledge of what is or is not allowed to be awaitable within that coroutine. For example, a generator<T> coroutine type is known to not allow any use of co_await within the body.

Secondly, there is no currently known implementation of a concept that can reliably answer this question for any combination of promise type and awaitable type without resorting to compiler intrinsics.

The main difficulty here centers around detecting whether or not the promise type contains an `await_transform` member.

The wording from N4760 8.3.8(3.2) states:

> … Otherwise, the *unqualified-id* `await_transform` is looked up within the scope of *P* by class member access lookup (6.4.5), and if this lookup finds **at least one declaration**, then *a* is `p.await_transform(`*`cast-expression`*`);` otherwise, *a* is the *cast-expression*.

The best approach so far for detecting whether there is at least one declaration of await_transform on the promise type is a technique suggested by Eric Niebler. This technique involves creating a class that multiply inherits from both the promise type and from another type that defines an await_transform member. If taking the address of the await_transform member of the derived class is ill-formed then this must be because the promise type also defines an await_transform member, making the identifier ambiguous in the derived class.

Example: Detecting the presence of await_transform in a promise type https://godbolt.org/z/50SVO3

```
struct __check_await_transform {
  void await_transform() {}
};

template <class Promise>
struct __check : __check_await_transform, Promise {};

template <class Promise>
concept HasAwaitTransform =
  !requires { &__check<Promise>::await_transform; };
```

The only limitation of this approach found so far is that it fails when the promise type is marked as final. However, it is not uncommon[4] to mark a promise_type as final to limit the risk of accidentally introducing UB if a type inherits from the promise_type and forgets to override get_return_object() to construct a coroutine_handle using the most-derived promise type.

[4] https://github.com/lewissbaker/cppcoro/blob/fc76dadcd058a6d74d3c6586eb4973921e226b49/include/cppcoro/task.hpp#L122

A compiler intrinsic may be necessary to reliably detect the presence of the `await_transform()` method in all cases. Alternatively, it may be possible to use the proposed features from the Reflection TS to detect the existence of the `await_transform()` member. This is an area of future research.

Wording

Formal wording can be provided pending an initial review of the concepts.

For now, there is a definition of the proposed concepts and metafunctions in the Reference Implementation section which can be used in place of formal wording.

Acknowledgements

Thanks to Eric Niebler for assistance with the concept definitions.
And thanks to Gor Nishanov for providing feedback on early drafts.

Appendix A - Reference Implementation

See reference implementation under Compiler Explorer: https://godbolt.org/z/9dapP6

```
// <experimental/coroutine>

#include <type_traits>
#include <concepts>

namespace std::experimental
{
  template<typename _Tp>
  struct __is_valid_await_suspend_return_type : false_type {};

  template<>
  struct __is_valid_await_suspend_return_type<bool> : true_type {};

  template<>
  struct __is_valid_await_suspend_return_type<void> : true_type {};

  template<typename _Promise>
  struct __is_valid_await_suspend_return_type<coroutine_handle<_Promise>>
  : true_type {};
  template<typename _Tp>
  concept _AwaitSuspendReturnType =
    __is_valid_await_suspend_return_type<_Tp>::value;

  template<typename _Tp>
  concept Awaiter =
    requires(_Tp&& __awaiter, coroutine_handle<void> __h)
    {
      // await_ready() result must be contextually convertible to bool.
      __awaiter.await_ready() ? void() : void();
      __awaiter.await_suspend(__h);
      requires _AwaitSuspendReturnType<decltype(
        __awaiter.await_suspend(__h))>;
      __awaiter.await_resume();
    };

  template<typename _Tp, typename _Result>
  concept AwaiterOf =
    Awaiter<_Tp> &&
    requires(_Tp&& __awaiter)
    {
      { __awaiter.await_resume() } -> _Result;
    };

  template<typename _Tp>
  concept _WeakHasMemberCoAwait =
    requires(_Tp&& __awaitable)
    {
      static_cast<_Tp&&>(__awaitable).operator co_await();
    };

  template<typename _Tp>
  concept _WeakHasNonMemberCoAwait =
    requires(_Tp&& __awaitable)
    {
      operator co_await(static_cast<_Tp&&>(__awaitable));
    };

  template<_WeakHasMemberCoAwait _Tp>
  decltype(auto) get_awaiter(_Tp&& __awaitable)
    noexcept(noexcept(static_cast<_Tp&&>(__awaitable).operator
co_await()))
  {
    return static_cast<_Tp&&>(__awaitable).operator co_await();
  }

  template<_WeakHasNonMemberCoAwait _Tp>
  decltype(auto) get_awaiter(_Tp&& __awaitable)
    noexcept(noexcept(operator co_await(static_cast<_Tp&&>(__awaitable))))
  {
    return operator co_await(static_cast<_Tp&&>(__awaitable));
  }

  template<typename _Tp>
    requires !_WeakHasNonMemberCoAwait<_Tp> && !_WeakHasMemberCoAwait<_Tp>
  _Tp&& get_awaiter(_Tp&& __awaitable) noexcept
  {
    return static_cast<_Tp&&>(__awaitable);
  }

  template<typename _Tp>
  struct awaiter_type
  {
    using type = decltype(
      std::experimental::get_awaiter(std::declval<_Tp>()));
  };

  template<typename _Tp>
  using awaiter_type_t = typename awaiter_type<_Tp>::type;

  template<typename _Tp>
  concept Awaitable =
    Movable<_Tp> &&
    requires(_Tp&& __awaitable)
    {
      { std::experimental::get_awaiter(static_cast<_Tp&&>(__awaitable)) }
        -> Awaiter;
    };

  template<typename _Tp, typename _Result>
  concept AwaitableOf =
    Awaitable<_Tp> &&
    requires(_Tp&& __awaitable)
    {
      { std::experimental::get_awaiter(static_cast<_Tp&&>(__awaitable)) }
        -> AwaiterOf<_Result>;
    };

  template<typename _Tp>
  struct await_result {};

  template<Awaitable _Tp>
  struct await_result<_Tp>
  {
    using type = decltype(
      std::declval<awaiter_type_t<_Tp>&>().await_resume());
  };

  template<typename _Tp>
  using await_result_t = typename await_result<_Tp>::type;
}
```

Usage examples:

```
#include <experimental/task>
#include <experimental/coroutine>

using namespace std;
using namespace std::experimental;

template<Awaitable A, Awaitable B>
task<await_result_t<B>> sequence(A a, B b)
{
  co_await std::move(a);
  co_return co_await std::move(b);
}

template<Awaitable A, Invocable<await_result_t<A>> Func>
struct transform_awaitable
{
  A awaitable;
  Func func;

  auto operator co_await() &&
  {
    struct awaiter
    {
      awaiter_type_t<A> inner;
```

```
        Func&& func;

        awaiter(A&& awaitable, Func&& func)
        : inner(std::experimental::get_awaiter(static_cast<A&&>(awaitable)))
        , func(static_cast<Func&&>(func))
        {}

       decltype(auto) await_ready()
       {
         return inner.await_ready();
       }

       template<typename Handle>
       auto await_suspend(Handle h) -> decltype(inner.await_suspend(h))
       {
         return inner.await_suspend(h);
       }

       decltype(auto) await_resume()
       {
         return std::invoke(
           static_cast<Func&&>(func), inner.await_resume());
       }
     };

     return awaiter{static_cast<A&&>(awaitable),
static_cast<Func&&>(func)};
   }

   auto operator co_await() &
     requires Awaitable<A&> && Invocable<Func&, await_result_t<A&>>
   {
     struct awaiter
     {
       awaiter_type_t<A&> inner;
       Func& func;

       awaiter(A& awaitable, Func& func)
       : inner(std::experimental::get_awaiter(awaitable))
       , func(func)
       {}

       decltype(auto) await_ready() { return inner.await_ready(); }
```

```
      template<typename Handle>
      auto await_suspend(Handle handle)
        -> decltype(inner.await_suspend(handle))
      {
        return inner.await_suspend(handle);
      }

      decltype(auto) await_resume()
      {
        return std::invoke(func, inner.await_resume());
      }
    };

    return awaiter{ awaitable, func };
  }
};
```