Document No.	P1713R0
Date	2019-06-13
Reply To	Lewis Baker <lbaker@fb.com>
Audience	Evolution
Targeting	C++20

Allowing both `co_return;` and `co_return value;` in the same coroutine

Introduction

In the C++ working draft (N4810) section [dcl.fct.def.coroutine]/6 currently specifies that:

> The *unqualified-id*s `return_void` and `return_value` are looked up in the scope of the promise type. If both are found, the program is ill-formed.

I believe this restriction is unnecessary to place on coroutine promise types and prevents some interesting use cases. In particular, this restriction makes syntax for implementing tail-recursion of coroutines unnecessarily cumbersome.

It also forces library authors writing generic coroutine types to specialise their promise_type for the void-return-type case, even when the only difference between the implementations is whether it has a `return_void()` or a `return_value()` method. Implementations could be greatly simplified if they could conditionally enable either the `return_void()` or `return_value()` method by adding an appropriate requires clause these methods within a generic `promise_type` definition.

I propose that we amend the specification to allow defining coroutine promise types that define both the `return_void` and `return_value` members by removing the entirety of paragraph 6.

Background

The original motivation for this restriction, as far as I understand it, is to maintain consistency with normal functions. A normal function cannot contain both `return;` and `return someValue;` statements.

This rule makes perfect sense for normal functions. The type of the expression passed to the `return` statement in a function directly corresponds to the return-type of that function. A function cannot have both a `void` and non-`void` return-type at the same time so it cannot make sense to have both `return;` and `return someValue;` in the same function.

For example, the following function body isn't able to fulfill the contract of the function signature with the second `return` statement (it needs to return an `int`) and so the program is considered ill-formed.

```
int f()
{
  if (someCond) return 123;
  else return; // Error: f() needs to return an 'int' value.
}
```

However, with coroutines, the return-type of a coroutine is ***not*** directly tied to the type passed to the `co_return` statement. The author of the coroutine promise type is able to control the semantics of the `co_return` statement by defining either a `return_value` or `return_void` method on the promise type.

A common example of this is a coroutine that has a return-type of `generator<int>`.
This coroutine allows a `co_return;` statement, a statement that is equivalent to returning a `void` value, even though the coroutine's return-type is not `void`.

```
generator<int> collatz_sequence(int n)
{
  while (true)
  {
    co_yield n;

    // Returning 'void' even though coroutine return-type is generator<int>.
    if (n == 1) co_return;

    if (n % 2 == 0) n /= 2;
    else n = (3 * n + 1) / 2;
  }
}
```

With the current wording a coroutine promise type author is also able to define multiple overloads of `return_value()`, allowing the coroutine to define different semantics for `co_return` statements based on the expression type passed.

It does not seem like a big step to go from allowing the user to define different semantics for two different types passed to `co_return <expr>` by defining two overloads of `return_value()` to allowing the user to define different semantics for `co_return` and `co_return <expr>` by defining both `return_void()` and `return_value()`.

Tail-Recursion of Coroutines

One of the motivating use-cases for the proposed change is to allow use of the `co_return` keyword as a convenient and intuitive syntax for indicating a tail-recursive call to a coroutine.

A tail-recursive call to another coroutine is an operation where the calling coroutine frame is destroyed before resuming execution of the coroutine that is being called in the tail-position. This allows you to perform recursion in the tail-position to an arbitrary recursion depth while needing at most two coroutine frames to be allocated at any one time.

Example: A simple tail-call of an async task.

```
recursive_task<T> bar();

recursive_task<T> foo_no_tail_recursion()
{
  co_await do_something();

  // Call bar() and await result, unwrapping task<T> to obtain T value.
  // Then returns the value of type T.
  //  - calls return_value(T) if T is non-void.
  //  - calls return_void() if T is void.
  co_return co_await bar();
}

recursive_task<T> foo_tail_recursive()
{
  co_await do_something();

  // Return recursive_task<T> value directly as way of indicating that
  // the result of bar()'s task should be used as the result
  // of foo_tail_recursive().
  co_return bar();
```

```
}

task<> usage()
{
  recursive_task<T> t = foo_tail_recursive();
  T result = co_await t;
}
```

The semantics of the tail-recursive statement `co_return bar();` is as follows:

- Call `bar()` to create a coroutine frame for `bar()`.
  This coroutine will immediately suspend at `initial_suspend` point and return the `recursive_task<T>` RAII object.
- Transfer ownership of `bar()`'s coroutine frame to the `recursive_task<T>` object that currently owns the `foo_tail_recursive()` coroutine frame.
  In the example above, this means updating the coroutine handle stored in the variable `` `t` `` in the `usage()` coroutine.
- Transfer responsibility for resuming awaiter of `foo_tail_recursive()`'s task object to `bar()`'s promise object.
  In the example above, the awaiting coroutine is `usage()` so this would be copying the `coroutine_handle` for the `usage()` coroutine into `bar()`'s promise object.
- Destroy `foo_tail_recursive()` coroutine frame.
- Resume execution of `bar()`.

Note: To guarantee that these tail-recursive statements have bounded memory usage of both heap-allocated coroutine frames and stack-frames, it makes use of the symmetric transfer[1] facility when suspending one coroutine and resuming another.

It is already possible to implement such a tail-recursive call operation for tasks that return a value. This is because we can define two overloads of `return_value()`: one taking a `T` and one taking a `recursive_task<T>&&`.

However, it is not currently possible to do this for `void`-returning tasks, as that would require defining `return_void()` for the normal return case and `return_value(recursive_task<void>&&)` to handle the tail-recursive call case. Something that is currently disallowed by the current coroutines wording.

If we allowed promise types to define both `return_void` and `return_value` then this would allow implementation of the tail-call capabilities of `recursive_task` for all types, not just non-`void` types.

[1] P0913R0 - "Add symmetric coroutine control transfer"

Tail-Recursive Generators

Another use-case for coroutine tail-recursion is with generator coroutines.

The cppcoro library provides a recursive_generator<T> type that allows a coroutine to lazily produce a sequence of values of type, `T`, by `co_yield`-ing either a value of type `T` or a `recursive_generator<T>` value.

If a `recursive_generator<T>` value is yielded then this is equivalent to yielding all of the values produced by that generator. Execution of the current coroutine only resumes once all of the nested generator values have been consumed.

The advantage of this approach is that it allows the consumer to directly resume the leaf-most nested coroutine to produce the next element inside its call to `iterator::operator++()` rather than requiring O(depth) resume/suspend operations that would be required if using a non-recursive `generator`.

For example: Traversing values of a binary tree in left-node-right order.

```
// Given a basic tree structure
template<typename T>
struct tree
{
  tree<T>* left;
  tree<T>* right;
  T value;
};

recursive_generator<T> traverse_tree(tree<T>* t)
{
  if (t->left) co_yield traverse_tree(t->left);
  co_yield t->value;
  if (t->right) co_yield traverse_tree(t->right);
}

void usage()
{
  tree<std::string>* root = get_a_tree();
  for (std::string& value : traverse_tree(root))
  {
    std::cout << value << std::endl;
```

```
  }
}
```

Notice in the `traverse_tree()` coroutine that there is no logic in the coroutine body after the `co_yield traverse_tree(t->right);` statement. The coroutine has suspended execution, but the only thing the continuation is going to do when it resumes is run to completion, suspend at `co_await promise.final_suspend()` and then resume its parent coroutine or return to the consumer.

When execution returns to the parent/consumer after the generator has run to completion the parent/consumer will then typically immediately destroy the generator object, which will in turn destroy the coroutine frame, freeing its memory.

However, this means that we are deferring releasing the memory used by the parent coroutine frame until the consumer has finished consuming all of the child generator's elements even though the parent coroutine is not going to be producing any more values.

If we could instead make use of tail-recursion in the case where a `co_yield` statement occurs in the tail position then this would allow the parent coroutine frame to be freed earlier, before resuming execution of the nested coroutine.

Doing so would allow `recursive_generator` coroutines to support recursion to an arbitrary depth when recursing in the tail position. This can be done using only a bounded amount of memory for the coroutine frames; typically at most two coroutine frames allocated at any one time.

Tail call syntax

My preferred syntax for indicating a tail-recursive yield is to allow `co_return std::move(childGenerator);` in the place of `co_yield childGenerator;`.

For example, the above `traverse_tree()` would become:

```
recursive_generator<T> traverse_tree_tail_recursive(tree<T>* t)
{
  if (t->left) co_yield traverse_tree_tail_recursive(t->left);
  co_yield t->value;
  // Use of co_return indicates tail-recursion on right subtree
  if (t->right) co_return traverse_tree_tail_recursive(t->right);
}
```

The problem with this approach is that it requires the `promise_type` for the coroutine to have both a `return_void()` method (for the case where execution runs off the end of the coroutine indicating the end of the range) and a `return_value(recursive_generator<T>&&)` method (for the case where we are performing tail-recursion). This is something which is currently banned by the wording in N4810 [dcl.fct.def.coroutine]/6.

Alternative tail call syntax

There are alternative syntaxes that could be implemented while staying within the current wording. However, these alternative syntaxes have downsides.

Alternative Syntax 1: Overload `co_yield` with a `tail_call()` helper function.

```
recursive_generator<T> traverse_tree_alternative1(tree<T>* t)
{
  if (t->left) co_yield traverse_tree_alternative1(t->left);
  co_yield t->value;
  if (t->right) co_yield tail_call(traverse_tree_alternative1(t->right));
}
```

This would work by having the `tail_call()` helper wrap the `recursive_generator<T>` object in a new type, say `recursive_generator_tail_call<T>` and then providing a `yield_value()` overload for that type which could then perform the tail-recursion operation.

This has the downside of requiring the more verbose syntax.

It is also less obvious to the developer that the coroutine will not continue exection after executing the `co_yield tail_call(...)` expression.

It may also be difficult for the compiler to determine that execution does not continue after the `co_yield tail_call(...)` expression which could make it more difficult to issue warnings about dead-code, etc. that would otherwise be possible were we using `co_return`.

Alternative 2: Remove `return_void()` and keep only `return_value(recursive_generator<T>&&)`

```
recursive_generator<T> traverse_tree_alternative2(tree<T>* t)
{
  if (t->left) co_yield traverse_tree_alternative2(t->left);
  co_yield t->value;
```

```
  if (t->right) co_return traverse_tree_alternative2(t->left);

  // Execution not allowed to run-off end.
  // Return a sentinel value instead.
  co_return recursive_generator<T>{};
}
```

This approach works by using a special sentinel value (in this case a default-constructed `recursive_generator<T>` value) that can be returned to indicate that no tail-recursion should be performed.

This has the downside of making simple generator coroutines more verbose as it forces every `recursive_generator` coroutine to have a `co_return` statement, not just the ones that make use of tail-recursion. A generator coroutine can no longer just let execution run off the end since that is undefined-behaviour if the promise object has no `return_void()` method.

Support returning error-types from a task

The opensource folly library has a folly::Try<T> type, which represents either a value or an exception, which is returned from some APIs.

If we want to return a `folly::Try<T>` value as the return-value of a folly::coro::Task<T> coroutine then we can extract the value by calling the .value() method and this will either return a reference to the value or will rethrow the exception.

However, in the interests of avoiding rethrowing an exception just to have it caught again and the exception_ptr recaptured by `promise.unhandled_exception()` we can add support for returning the `folly::Try<T>` value directly by overloading the `promise.return_value()` method for `folly::Try<T>` and have the coroutine copy/move the `exception_ptr` value directly into the promise if required rather than rethrowing the exception.

This can be implemented for all `Task<T>` types where `T` is not `void`. However, for the `void` case we cannot implement this facility because we cannot support both '`co_return;`' and '`co_return result;`' in the same coroutine.

Reducing boilerplate in coroutine promise_types

When defining generic coroutine task types that need to support any type T we need to define a promise_type for each type T.

With the current design, it is not allowed to define both return_value and return_void on the same promise_type, even if one of the methods is deleted.

This means that in order to support both void and non-void types we end up needing to specialise the `promise_type` for the void case to give it a `return_void()` method and then have a generic implementation for the non-void types which has a `return_value()` method.

If we adopt the change proposed by this paper then we can write a single generic promise_type and control the availability of `co_return;` vs `co_return value;` by adding appropriate requires-clauses to the `return_void()` and `return_value()` methods.

For example: Defininig a single generic `promise_type` is possible with the proposed change

```
template<typename T>
class task {
public:
  class promise_type {
    struct final_awaiter { ... };

    enum class state { empty, value, error };
    state state_ = state::empty;
    std::continuation_handle<> consumer_;
    union {
      manual_lifetime<T> value_;
      manual_lifetime<exception_ptr> error_;
    };

  public:
    promise_type() {}
    ~promise_type() {
      switch (state_) {
        case state::empty: break;
        case state::value: value_.destruct(); break;
        case state::error: error_.destruct(); break;
      }
    }

    // All common implementation.
    task get_return_object();
    suspend_always initial_suspend();
    final_awaiter final_suspend();
    void unhandled_exception();

    decltype(auto) value() {
      if (state_ == state::error)
        std::rethrow_exception(error_.get());
      return std::move(value_).get();
    }

    template<ConvertibleTo<T> U>
      requires (!std::is_void_v<T>)
    void return_value(U&& value) {
      value_.construct((U&&)value);
      state_ = state::value;
    }

    void return_void() noexcept requires std::is_void_v<T> {
      value_.construct(); // no-op
      state_ = state::value;
```

```
    }
  };

  // ... etc for the rest of the task definition common to all T types.
};
```

Final Words

There are many cases where the restriction to disallow both `return_void()` and `return_value()` methods being defined on the same promise_type has prevented valid use-cases or required extra boiler-plate to handle both void and non-void return-types.

As far as I was able to discover, the original reason for disallowing both 'co_return;' and 'co_return value;' in the same coroutine was to be consistent with ordinary functions, which disallow both 'return;' and 'return value;' in the same function. However, the reasons for disallowing this in ordinary functions do not hold for coroutines because the library is able to customise the behaviour of co_return to give it appropriate semantics for the current coroutine type.

Lifting this restriction by striking [dcl.fct.def.coroutine]/6 would allow more use cases and simplify code.