Paper.	P1477R0
Audience	Evolution
Author	Lewis Baker <lbaker@fb.com>
Date	2019-01-20

Coroutines TS Simplifications

Abstract

The paper P0973R0 raised a concern about the perceived size and complexity of the language/library interaction and the number of customisation points defined by the Coroutines TS.

The paper P1342R0 lists some potential simplifications we could make to the interface defined by the Coroutines TS that would reduce the number of customisation points and simplify some of the rules for other customisation points.

This paper explains in more detail some of the proposed simplifications from P1342R0.

The simplifications proposed by this paper are:

- Merge initial_suspend() into get_return_object()
- Simplify final_suspend() to accept and return a coroutine_handle instead of returning an awaitable object.
- Rename await_transform() to make naming consistent with other methods

Note that all of these simplifications are functionality-preserving and the net result of these changes are to reduce the amount of code required to implement coroutine promise_types. However, this also means that each of these changes is a breaking change for existing code written against the Coroutines TS.

Implementation of these simplifications in Clang was not yet complete in time for the mailing deadline.

Simplifying initial_suspend() and get_return_object()

The proposed change is to simplify the semantics of the coroutine startup by:

- Removing the need to define an initial_suspend() method
- Specifying that the coroutine frame is always created in a suspended state.
- Modifying the call to promise.get_return_object() to pass the initial coroutine handle as a parameter.

```
// Coroutines TS promise_type interface
struct promise_type
{
  T get_return_object();

  Awaitable<void> initial_suspend();
  ...
};
```

```
// Proposed TS promise_type interface
struct promise_type
{
  T get_return_object(coroutine_handle<promise_type> h);
  ...
};
```

Reducing boiler-plate in initial_suspend() and get_return_object()

The `initial_suspend()` method is currently required to return an awaitable object. Most implementations typically return `std::suspend_never` (if the coroutine should start executing immediately) or `std::suspend_always` (if the coroutine should *not* start executing immediately). However, there are some use-cases where the coroutine may want to conditionally start executing immediately – eg. an actor-model task that immediately starts executing if no other methods are currently executing on the actor and otherwise suspends and enqueues itself onto a list of pending calls. In these cases, the `promise_type` must define a custom awaiter type with implementations of the `await_ready()`, `await_suspend()` and `await_resume()` methods.

The result of the call to await_resume() of the initial_suspend() awaitable is always discarded and in all known coroutine-types has an empty body. Having to define this empty method for non-trivial initial_suspend() methods adds to the boiler-plate needed to implement a promise_type.

By creating the coroutine in an initially suspended state and passing the coroutine_handle of the suspended coroutine to get_return_object() this allows the decision of whether or not to immediately launch the coroutine or defer its launch to be implemented inline in get_return_object() without needing to define a custom Awaiter type.

RAII objects and exception-safety

A common pattern is for the get_return_object() method to return a RAII object that takes ownership of the lifetime of the coroutine frame and is responsible for calling .destroy() on the coroutine_handle. There are a few minor issues that result from this.

As the coroutine_handle for the current coroutine is not provided to the `get_return_object()` method, implementations will typically need to call the static factory function `std::coroutine_handle<promise_type>::from_promise(*this)` to reconstruct the `coroutine_handle` from the promise object. Other than being a verbose way of getting hold of the coroutine handle (an intentional design decision) the promise_type author needs to be careful what they do with the handle as the coroutine is not yet suspended and so it is undefined behaviour to call `.resume()` or `.destroy()`.

If we pass this `coroutine_handle` to a RAII object its destructor will typically call `.destroy()` on the handle. So if the RAII object were to be destructed before the coroutine reached the initial-suspend-point then we could end up deleting a coroutine that was not yet suspended – leading to undefined-behaviour. If the RAII object is returned from get_return_object() and then an unhandled exception is thrown from the `initial_suspend()` call then the coroutine frame will be implicitly destroyed by compiler-generated code in additon to the RAII object destructor attempting to destroy the coroutine frame, leading to a double free.

Example: A thread-pool coroutine type with a subtle double-free bug

```
struct tp_task {
  struct promise_type {
    tp_task get_return_object() {
      return tp_task{ std::coroutine_handle<promise_type>::from_promise(*this) };
    }

    auto initial_suspend() {
      struct awaiter {
        static void CALLBACK callback(PTP_CALLBACK_INSTANCE instance, void* data) {
          coroutine_handle<promise_type>::from_address(data).resume();
        }
        bool await_ready() { return false; }
        void await_suspend(std::coroutine_handle<promise_type> h) {
          // Use Windows Thread Pool API to schedule resumption onto thread pool.
          if (!TrySubmitThreadpoolCallback(&awaiter::callback, h.address(), nullptr)) {
            throw std::system_error{(int)GetLastError(), std::system_category()};
          }
        }
        void await_resume() {}
      };
      return awaiter{};
    }
    ...
  };

  std::coroutine_handle<promise_type> coro;

  explicit tp_task(std::coroutine_handle<promise_type> h) : coro(h) {}

  ~tp_task() { if (coro) coro.destroy(); }
  ...
};
```

In the above example, the tp_task object is returned from get_return_object() and then 'co_await p.initial_suspend()' is evaluated. However, if it fails to schedule the coroutine onto the thread-pool then an exception is thrown and this will propagate back out to the caller of the coroutine, destroying the coroutine frame and also destroying the tp_task object returned from get_return_object(), which then also tries to destroy the coroutine frame.

Issue #24 captured in the Coroutines TS Issues paper (P0664) discusses placing all or part of the 'co_await p.initial_suspend()' expression inside the implicit try/catch around the coroutine body. This would cause the exception to be caught and processed by p.unhandled_exception() instead of it propagating out to the caller.

Without this resolution to issue #24 we would instead need to return a proxy object that was implicitly convertible to tp_task but that does not call .destroy() in the destructor. Thus we only transfer ownership of the handle to the RAII object if the '`co_await p.initial_suspend()`' expression does not throw and so avoid double-deletion of the frame if it does throw.

With the simplifications proposed in this paper this task type can be implemented more simply and safely:

```
template<typename T>
struct tp_task {
  struct promise_type {
    static void CALLBACK callback(PTP_CALLBACK_INSTANCE instance, void* data) {
      coroutine_handle<promise_type>::from_address(data).resume();
    }

    tp_task get_return_object(std::coroutine_handle<promise_type> h) {
      // Use Windows Thread Pool API to schedule resumption onto thread pool.
      if (!TrySubmitThreadpoolCallback(&promise_type::callback, h.address(), nullptr)) {
        throw std::system_error{(int)GetLastError(), std::system_category()};
      }
      // Only construct RAII object once we know we will complete successfully.
      return tp_task{h};
    }
    ...
  };

  std::coroutine_handle<promise_type> coro;

  tp_task(std::coroutine_handle<promise_type> h) : coro(h) {}

  ~tp_task() { if (coro) coro.destroy(); }
  ...
};
```

There is an outstanding question of whether we should define the semantics of get_return_object() such that if an exception is thrown from get_return_object() that the coroutine is not implicitly destroyed. ie. that the call passes ownership to the promise_type.

This would allow the coroutine creation to be implemented as:

```
task<T> some_function(int arg)
{
  auto* frame = new __frame{arg};
  return frame->promise.get_return_object(
    coroutine_handle<promise_type>::from_promise(frame->promise));
}
```

Examples of merged get_return_object() and initial_suspend()

Existing Coroutines TS

```cpp
// Lazily-started task<T> promise_type
struct promise_type {
  task<T> get_return_object() {
    return task<T>{
      std::coroutine_handle<promise_type>::
        from_promise(*this) };
  }
  std::suspend_always initial_suspend() {
    return {};
  }
  ...
};
```

With the proposed changes

```cpp
struct promise_type {
  template<typename Handle>
  task<T> get_return_object(Handle coro) {
    return task<T>{ coro };
  }
  ...
};
```

Existing Coroutines TS

```cpp
// Eager, oneway_task promise_type
struct promise_type {
  oneway_task get_return_object() {
    return {};
  }
  std::suspend_never initial_suspend() {
    return {};
  }
  ...
};
```

With the proposed changes

```cpp
struct promise_type {
  template<typename Handle>
  oneway_task get_return_object(Handle coro) {
    coro.resume(); // Start coroutine
    return {};
  }
  ...
};
```

Existing Coroutines TS

```cpp
// std::optional promise_type
struct promise_type {
  std::optional<T>* result = nullptr;

  struct optional_proxy {
    std::optional<T> result;
    optional_proxy(promise_type* p) {
      p->result = &result;
    }
    operator std::optional<T>() && {
      return std::move(result);
    }
  };

  optional_proxy get_return_object() {
    return optional_proxy{ this };
  }

  std::suspend_never initial_suspend() {
    return {};
  }

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

  template<typename U>
  auto await_transform(const std::optional<U>& value) {
    struct awaiter {
      const std::optional<U>& value;
      bool await_ready() { return value.has_value(); }
      void await_suspend(coroutine_handle<>) {}
      const T& await_resume() { return *value; }
    };
    return awaiter{value};
  }

  ...
};
```

With the proposed changes

```cpp
struct promise_type {
  std::optional<T>* result = nullptr;

  std::optional<T> get_return_object(
      coroutine_handle<promise_type> h) {
    scope_guard g{[h] { h.destroy(); }};
    std::optional<T> result;
    h.promise().result = &result;
    h.resume(); // Start coroutine
    return result; // Permits NRVO
  }

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

  template<typename U>
  auto await_transform(const std::optional<U>& value) {
    struct awaiter {
      const std::optional<U>& value;
      bool await_ready() { return value.has_value(); }
      void await_suspend(coroutine_handle<>) {}
      const T& await_resume() { return *value; }
    };
    return awaiter{value};
  }

  ...
};
```

Simplifying final_suspend()

The proposed change is to:

- Change final_suspend() from
  - a method taking no arguments and returning an Awaitable type to;
  - a method taking a coroutine_handle and returning coroutine_handle.
    ie. the equivalent of the await_suspend() method of the awaitable returned from final_suspend() under the current Coroutines TS design.
- No longer implicitly destroy of the coroutine frame if execution runs off the end of the coroutine.

Changing the signature of final_suspend()

The current specification of the Coroutines TS requires the final_suspend() method to return an Awaitable object. The compiler will insert the statement 'co_await promise.final_suspend()' at the end of the coroutine body.

For non-async coroutines the final_suspend() method typically returns std::suspend_always so that execution suspends and returns to the caller. For detached/one-way tasks that do not have a continuation and that must self-destroy they will typically return std::suspend_never which means the coroutine will not suspend at the final-suspend point and will continue to run to completion and implicitly destroy the coroutine frame before then returning execution to the caller/resumer.

However, for most asynchronous coroutine-types this Awaitable needs to suspend the current coroutine and resume its continuation. As this is something that is promise_type-specific this means that each promise_type will generally need to implement its own Awaitable object.

For example, the implementation for a lazy task<T> type (see P1056) would typically have a final_suspend() method that looks like this:

```
template<typename T>
struct task {
  struct promise_type {
    std::coroutine_handle<> continuation;
    std::variant<std::monostate, T, std::exception_ptr> result;
    ...
    auto final_suspend() noexcept {
      struct awaiter {
        bool await_ready() noexcept { return false; }
        auto await_suspend(std::coroutine_handle<promise_type> h) noexcept {
          return h.promise().continuation;
        }
        void await_resume() noexcept {}
      };
      return awaiter{};
    }
  };
  ...
};
```

Part of the reason why the final-suspend point was defined in terms of co_await was for uniformity of specification and symmetry with the initial_suspend() method. It is relatively straight-forward to explain and teach that a coroutine simply has an implicit ‘co_await promise.initial_suspend();’ at the open brace and an implicit ‘co_await promise.final_suspend();’ at the close brace.

However, the final-suspend point is special. It is not permitted to call `.resume()` a coroutine suspended at the final-suspend point - you can only call `.destroy()`. This means that you cannot typically just reuse arbitrary Awaitable types (std::suspend_never/suspend_always are exceptions since they do not call .resume() on the coroutine_handle).

If the proposed removal of initial_suspend() is adopted then the motivation for maintaining symmetry with initial_suspend() is no longer present.

Also, a co_await expression can potentially have a result. The await_resume() method is called to produce the value of a co_await expression. However, the result of the ‘co_await promise.final_suspend()’ method is always discarded and so in all practical implementations of final_suspend(), the await_resume() method returns void and has an empty body. It is only ever executed if the coroutine does not suspend at the final-suspend point.

The proposed change simplifies the final_suspend() method to the essential component. ie. the await_suspend() method.

With Coroutines TS	With proposed changes

`// Always suspend – eg. generator<T>::promise_type`
`std::suspend_always final_suspend() {`
`  return {};`
`}`
	`auto final_suspend(std::coroutine_handle<promise_type>) {`
	`  return std::noop_coroutine();`
	`}`

`// Never suspend – eg. detached_task::promise_type`
`std::suspend_never final_suspend() {`
`  return {};`
`}`
	`auto final_suspend(std::coroutine_handle<promise_type> h) {`
	`  h.destroy();`
	`  return std::noop_coroutine();`
	`}`

`// Execute continuation. eg. task<T>::promise_type`
`auto final_suspend() {`
`  struct awaiter {`
`    bool await_ready() { return false; }`
`    auto await_suspend(`
`        std::coroutine_handle<promise_type> h) {`
`      return h.promise().continuation;`
`    }`
`    void await_resume() {}`
`  };`
`  return awaiter{};`
`}`
	`auto final_suspend(std::coroutine_handle<promise_type>) {`
	`  return this->continuation;`
	`}`

<pre>// Conditionally execute continuation or destroy
// eg. eager_task<T>::promise_type
auto final_suspend() {
  struct awaiter {
    bool await_ready() { return false; }
    std::coroutine_handle<> await_suspend(
        std::coroutine_handle<promise_type> h) {
      auto oldState =
        h.promise().state.exchange(state::finished);
      if (oldState == state::awaiting_coroutine) {
        return h.promise().continuation;
      } else if (oldState == state::detached) {
        h.destroy();
        return std::noop_coroutine();
      } else {
        assert(oldState == state::started);
        return std::noop_coroutine();
      }
    }
    void await_resume() {}
  };
  return awaiter{};
}</pre>
	<pre>std::coroutine_handle<> final_suspend(
	    std::coroutine_handle<promise_type> h) {
	  auto oldState = state.exchange(state::finished);
	  if (oldState == state::awaiting_coroutine) {
	    return continuation;
	  } else if (oldState == state::detached) {
	    h.destroy();
	    return std::noop_coroutine();
	  } else {
	    assert(oldState == state::started);
	    return std::noop_coroutine();
	  }
	}</pre>

Removing the implicit destroy when execution runs off the end

Under the Coroutines TS, if a coroutine does not suspend at the final suspend-point and execution runs off the end of the coroutine then the coroutine frame is implicitly destroyed.

The fact that we spell “destroy the current coroutine frame” as returning std::suspend_never from final_suspend() is subtle and can be surprising. If instead, we were to treat the coroutine frame as a resource that must always be explicitly destroyed the the code becomes easier to reason about because you can see the explicit call to `h.destroy()` in the `final_suspend()` method.

It can also be counter-productive efficiency-wise to destroy the coroutine-frame from within final_suspend(). The compiler is more likely to be able to apply the HALO optimisation[1] to elide the heap allocation of the coroutine frame if the `destroy()` method is called by the caller (eg. in the destructor of a RAII object) rather than in `final_suspend()`. This is because the compiler can more easily prove that the lifetime of the coroutine frame is strictly nested within the lifetime of the caller. Providing explicit special behaviour (implicit coroutine destroy) for something that is a performance anti-pattern seems counter-productive.

In N4557 11.4.4(9) the Coroutines TS states:

> The coroutine state is destroyed when control flows off the end of the coroutine or the destroy member function (21.11.2.4) of an object of type std::experimental::coroutine_handle associated with this coroutine is invoked.

The phrase “when control flows off the end of the coroutine” is unclear whether this applies to the case when execution exits the coroutine with an unhandled exception. The uncertainty here is captured as issue #25 in P0664R6 – C++ Coroutine TS Issues and has a proposed resolution that would require final_suspend(),

---

[1] See P0981R0 - Halo: coroutine Heap Allocation eLision Optimization: the joint response by Gor Nishanov and Richard Smith

operator co_await() and the await_ready(), await_suspend() and await_resume() methods to all be declared noexcept.

This proposed change to final_suspend() would be an alternative solution to issue #25. Namely that the coroutine is considered suspended at the final-suspend point before the call to final_suspend() and that any exceptions that propagate out of the call to final_suspend() will be rethrown to the caller/resumer and that the coroutine frame will not be implicitly destroyed. This would also make the behaviour of final_suspend() consistent with the behaviour of unhandled_exception() suggested in the proposed resolution for issue #25.

Making method names on promise_type consistent

This paper proposes renaming the await_transform() method on the promise_type to await_value() to improve consistency with the other method names on the promise_type interface.

The current interface defines:

- co_await <expr>' to map to 'co_await promise.await_transform(<expr>)' if there is any await_transform identifier found in the scope of the promise_type,
  otherwise it maps to `co_await <expr>'
- 'co_yield <expr>' to map to 'co_await promise.yield_value(<expr>)'
- 'co_return <expr>' to map to 'promise.return_value(<expr>)'

By renaming await_transform() to await_value() the naming of the methods becomes more consistent and therefore easier to teach. ie. the co_xxx keyword is translated into a call to the promise.xxx_value() method.

An alternative naming scheme suggested by Richard Smith was to name these methods as operators. I.e.

- 'co_yield <expr>' maps to 'co_await promise.operator co_yield(<expr>)'
- 'co_await <expr>' maps to 'co_await promise.operator co_await(<expr>)'
- 'co_return <expr>;' maps to 'promise.operator co_return(<expr>)'
- 'co_return;' maps to 'promise.operator co_return()'

Making await_transform()/await_value() mandatory

The other inconsistency between the co_await and co_yield/co_return keywords is that with the co_await keyword the await_transform() method is currently optional whereas yield_value() and return_value()/return_void() are mandatory to be able to use co_yield/co_return keyword within a coroutine.

This makes the co_await keyword supported by default in coroutines and a promise_type needs to explicitly declare await_transform() overloads as deleted to opt-out of supporting the co_await keyword (eg. like in generator<T>). Whereas a promise_type needs to explicitly opt-in to supporting the co_yield/co_return keywords by defining the yield_value/return_value/return_void methods.

The rationale here is that a `co_await` operand typically has a suitable default `operator co_await()` defined which can be forwarded to. Whereas a `co_yield`/`co_return` typically needs to interact with the `promise_type` and so can have no suitable default implementation.

If we were to define a user-authored co_await expression to always map to '`co_await promise.await_value(<expr>)`' then we would be making the rules for co_await consistent with the co_yield/co_return keywords. ie. that all keywords need to be explicitly opted-in to.

This would have the effect of async coroutine types, like task<T>, that did want to support co_await to needing to explicitly define an identity template `await_value()` method. It would also mean that coroutine types that did not want to support co_await, like generator<T>, would no longer need to explicitly declare deleted await_transform() methods on the promise_type.

Finally, the current rules for await_transform() make it difficult to build a template metafunction that can deduce what the type of a co_await expression will be within a given coroutine type. To be able to deduce the semantics of a co_await expression within a given coroutine type it is necessary to determine whether or not there is any await_transform() overloads defined on the promise_type. There is currently no known library solution that can reliably detect the presence of an await_transform identifier within a class in all situations.

By making await_transform() mandatory we can more simply define concepts that can check for the validity of a given call to the promise.await_transform() method.

Note that the alternative is to provide these queries as part of the standard library and require compiler magic to answer them. This compiler magic could later be replaced by the facilities proposed in the Reflection TS.

Examples of combined simplifications

Example: A lazy task<T> coroutine type

<table>
<tr><th>With current Coroutines TS</th><th>With the proposed changes</th></tr>
<tr><td><pre>
template<typename T>
struct task {
  struct promise_type {
    std::coroutine_handle<> continuation;
    std::variant<std::monostate, T, std::exception_ptr>
      result;

    task<T> get_return_object() {
      return task<T>{
        std::coroutine_handle<promise_type>::from_promise(
          *this) };
    }

    std::suspend_always initial_suspend() {
      return {};
    }

    auto final_suspend() {
      struct awaiter {
        bool await_ready() { return false; }
        auto await_suspend(
            std::coroutine_handle<promise_type> h) {
          return h.promise().continuation;
        }
        void await_resume() {}
      };
      return awaiter{};
    }

    void unhandled_exception() {
      result.template emplace<2>(
        std::current_exception());
    }

    template<ConvertibleTo<T> U>
    void return_value(U&& value) {
      result.template emplace<1>(static_cast<U&&>(value));
    }
  };
  ...
};
</pre></td><td><pre>
template<typename T>
struct task {
  struct promise_type {
    std::coroutine_handle<> continuation;
    std::variant<std::monostate, T, std::exception_ptr>
      result;

    task<T> get_return_object(
        coroutine_handle<promise_type> h) {
      return task<T>{ h };
    }

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

    void unhandled_exception() {
      result.template emplace<2>(
        std::current_exception());
    }

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

    template<ConvertibleTo<T> U>
    void return_value(U&& value) {
      result.template emplace<1>(static_cast<U&&>(value));
    }
  };
  ...
};
</pre></td></tr>
</table>

Example: A generator<T> implementation

With current Coroutines TS

```
template<typename T>
struct generator {
  struct promise_type {
    std::add_pointer_t<T> value;

    generator<T> get_return_object() {
      return generator<T>{
        std::coroutine_handle<promise_type>::from_promise(
          *this) };
    }
    std::suspend_always initial_suspend() {
      return {};
    }
    std::suspend_always final_suspend() {
      return {};
    }

    // Prevent use of 'co_await' within coroutine.
    template<typename U>
    void await_transform(U&& value) = delete;

    template<typename U>
    std::suspend_always yield_value(U&& value) {
      this->value = std::addressof(value);
      return {};
    }

    void return_void() {}

    void unhandled_exception() {
      throw;
    }
  };
  ...
};
```

With the proposed changes

```
template<typename T>
struct generator {
  struct promise_type {
    std::add_pointer_t<T> value;

    generator<T> get_return_object(
        coroutine_handle<promise_type> h) {
      return generator<T>{ h };
    }

    auto final_suspend(coroutine_handle<promise_type>) {
      return std::noop_coroutine();
    }

    template<typename U>
    std::suspend_always yield_value(U&& value) {
      this->value = std::addressof(value);
      return {};
    }

    void return_void() {}

    void unhandled_exception() {
      throw;
    }
  };
  ...
};
```

Example: A detached-task type

With current Coroutines TS

```
struct detached_task {
  struct promise_type {
    detached_task get_return_object() { return {}; }
    std::suspend_never initial_suspend() { return {}; }
    std::suspend_never final_suspned() { return {}; }
    void return_void() {}
    void unhandled_exception() { std::terminate(); }
  };
};
```

With the proposed changes

```
struct detached_task {
  struct promise_type {
    auto get_return_object(coroutine_handle<> h) {
      h.resume();
      return detached_task{};
    }
    auto final_suspend(coroutine_handle<> h) {
      h.destroy();
      return std::noop_coroutine();
    }
    void return_void() {}
    void unhandled_exception() { std::terminate(); }
  };
};
```

Proposed Wording Changes

A subsequent revision of this paper will provide proposed wording changes prior to the meeting in Kona.

Acknowledgements

Thanks to Eric Niebler and Gor Nishanov for feedback and input to this paper.