Coroutines: Use-cases and Trade-offs

Document Number: **P1493 R0** Date: 2019-02-19
Audience: WG21, EWG

Authors: Geoffrey Romer, Gor Nishanov, Lewis Baker, Mihail Mihailov

Abstract

The EWG chair has requested a paper from the authors of all coroutine proposals to analyze use cases and provide information which use-cases are supported better by one proposal over the other and what are the trade-offs.

Ville's Homework (Use Cases):

1) analysis of use cases, from user and/or library writer perspective.
2) What can proposal X do that proposal Y can't?
3) What are the trade-offs?

1. Overview

A coroutine is a generalization of a function: regular functions always start at the beginning and exit at the end, whereas coroutines can also suspend the execution to be resumed later at the point where they were left off.

A common implementation strategy for a coroutine is a compiler-based transformation of a function into a state machine. This paper examines several approaches to how to do this transformation and how the state machine can be exposed in the language. We will look at what are the usability and library writer trade-offs of various approaches.

The following is a quick reference to the papers discussing the proposals analyzed in this paper:

N4775:	Coroutines TS
P1063R2:	Core Coroutines
P1342R0:	Unifying Coroutines TS and Core Coroutines
P1362R0:	Incremental Approach: Coroutine TS + Core Coroutines
P1430R0:	First-class symmetric coroutines in C++

2. Coroutine Usability

The coroutine state machine object is by necessity an immovable object: it cannot be moved and it cannot be copied. A coroutine state machine object is often not useful by itself and requires wrapping into a class with higher level semantics. Usually, a type-erased wrapper that involves heap allocation and type erasure **or** an embedded wrapper which wraps the immutable coroutine objects into a semantically meaningful immovable class and in case of symmetric coroutines the behaviour is customized via subclassing the base coroutine class.

The following is an illustration of user facing syntax for both (using Core Coroutines syntax).

Type-erased coroutine

```
auto Traverse(BstNode<int> *node)
        [node][->]generator<string>
{
  if ( node == nullptr )
    return;

  [<-] Traverse ( node -> left );
  [<-] std::yield ( node -> value );
  [<-] Traverse ( node -> right );
}
```

Embedded coroutine (expected to be simpler in r3)

```
template <typename Range>
auto traverser(const Range& range) {
  using Sg = stack_generator<decltype(
    *begin(range)), void>;
  return Sg([&] {
    return [&] [->] Sg {
      for ( auto & element : range )
         [<-] std :: yield ( element );
    });
  });
}
```

While we expect that a future revision of Core Coroutines paper will bring embedded coroutine syntax closer to that of the type-erased coroutines, there are still usability challenges associated with the approach of embedding of the concrete coroutine state machine into the return type (or expose it as named class as in Symmetric Coroutines proposal).

Type-erased coroutines	Embedded coroutines
✓ Can be forward declared	✗ Cannot be forward declared
✓ Can be called recursively	✗ Cannot be called recursively[1]
✓ Can be put on ABI boundary	✗ Cannot be put on ABI boundary[2]
✓ Does not require solution to sizeof challenge	✗ Requires solution to a sizeof challenge
✓ Definition can reside in the implementation file	✗ Definition must reside in the header / module interface file
✓ Can be virtual	✗ Cannot be virtual
✗ BUT!!! Requires heap allocation & indirect calls	✓ Does not require heap allocation

[1] Symmetric Coroutine proposal allows self-recursion, but not mutual recursion at the moment.

[2] otherwise extremely fragile, as any change to implementation immediately leaks through an interface.

Tony Tables

Expected unwrapping (as of today)

Coroutines TS	Core Coroutines

`expected<int, Err> f(const string& s)`
`{`
`  int i = (co_await foo(s)).size();`
`  co_return i + co_await bar();`
`}`
	`auto f(const string& s)`
	`     [&] [->] expected<int, Err>`
	`{`
	`  int i = ([<-]foo(s)).size();`
	`  return i + [<-]bar();`
	`}`

Symmetric Coroutines =>	`expected<int, Err> foo() {`
	`  coroutine<expected<int, Err>(),`
	`            expected<int, Err>()>`
	`  f_coro(const string& s) {`
	`    int i = unwind_on_error()(foo(s)).size();`
	`    return i + unwind_on_error()(bar());`
	`  }`
	`  return f_coro()();`
	`}`

Note 1: Core coroutines library binding to support expected is much simpler than bindings for Coroutines TS today.

Note 2: Core coroutines guarantee no allocation by construction in this case, Coroutines TS offers no allocation as QoI. **Incremental path expects that Coroutines TS reaches full parity with Core Coroutines on this scenario.** See appendix for an example of library binding for the incremental path.

Type erased generators

Coroutines TS	Core Coroutines	Symmetric Coroutines

`gen<int> f() {`
`  co_yield 1;`
`  co_yield 2;`
`}`
`…`
`for (auto v: f())`
`  cout << v;`
	`auto f() [] [->] gen<int> {`
	`  [<-] std::yield(1);`
	`  [<-] std::yield(2);`
	`}`
	`…`
	`for (auto v: f())`
	`  cout << v;`
		`generator<int> f() {`
		`  yield(1);`
		`  yield(2);`
		`}`
		
		`unique_ptr<generator<int>> factory()`
		`{`
		`  return make_unique<f>();`
		`}`
		`…`
		`for (auto v: *(factory()))`
		`  cout << v;`

Note 1: Implementations of the Coroutines TS can apply Halo optimization to devirtualize, inline and remove heap allocations when the lifetime of a coroutine is enclosed in the lifetime of its caller (conditions under which this can be performed are listed in the HALO paper [P0981R0]). Doing similar optimization for library-based type-erase coroutines (as in Core Coroutines and Symmetric Coroutines) is more challenging and/or impossible for the cases where it is possible in Coroutines TS.

Note 2: Generators are safer to use under Coroutines TS and Core Coroutines since they allow to ban other kind of suspensions in the generators.

Type erased tasks (now)

Coroutines TS	**Core Coroutines**	**Symmetric Coroutines**

`task<int> f() {`
`  co_await g();`
`  ...`
`  co_return 42;`
`}`
	`No solution to tail resumption of the awaiting coroutine.`
	
	`No solution to exception propagation.`
		`No solution to exception propagation.`
		
		`detach mechanism is unsafe in multithreaded code`

Type erased tasks (future)

Coroutines TS	**Core Coroutines**	**Symmetric Coroutines**

`task<int> f(int n) {`
`  co_await g(n+1);`
`  ...`
`  co_return 42;`
`}`
	`task<int> f(int n) {`
	`  [<-] g(n+1);`
	`  ...`
	`  return 42;`
	`}`
		`task<int> f(int n) {`
		`  g(n+1)();`
		`  …`
		`  return 42;`
		`}`

Note 1: Addressing the aforementioned issues with Core Coroutines and Symmetric Coroutines may entail some additional API complexity ([P1362R0]).

Embedded (no-alloc) generators - the present

Coroutines TS	Core Coroutines	Symmetric Coroutines

`auto f(string s) {`
` return make_on_stack<64>(`
`   [=](auto) -> gen<string> {`
`     co_yield s;`
`   });`
`}`
	`auto f(string s) {`
	`  using Sg = stack_gen<string>;`
	`  return Sg([&] {`
	`    return [=] [->] Sg {`
	`      [<-] std::yield(s));`
	`    });`
	`  });`
	`}`
		`generator<string>`
		`f(string s) {`
		`  yield(s);`
		`}`
		
		`f f1;`

Unified Coroutines
 	
`auto f(string s) [->] stack_gen<string> {`
`  co_yield s;`
`}`		

Note 1; We expect that Core Coroutines and Incremental Coroutines TS will match Unified Coroutines syntax in the future.

Note 2: Embedded Coroutines of other types are also possible with no-alloc approach.

Note 3: The TS example requires an explicit estimate of the allocated size of the coroutine (`64`), and fail to build if the estimate is too low.

Embedded Coroutines (no alloc) - the future

Coroutines TS And Unified coroutines	Core Coroutines

`auto f(string s) [->] stack_gen<string> {`
`  co_yield s;`
`}`
	`auto f(string s) [=][->] stack_gen<string> {`
	`  [<-] std::yield(s);`
	`}`

Symmetric coroutine (same as before):
 	
`generator<string> f(string s) { yield(s); }`
`f f1;`	

We expected that user-facing experience and efficiency will be comparable under all proposals (that follow late-split implementation).

Safety and usability

All coroutines proposal except for Symmetric Coroutines provide tools for library designer to control coroutine experience for the end user. For example, generator coroutines can ban suspends due to awaiting on a future or unwrapping of an expected. Coroutines returning expected, can ban awaiting on asynchronous expression and yielding values, etc.

Coroutines TS	Symmetric Coroutines

future<void> do_async();
 	
generator<int> g() {
  co_yield 42;
  co_await do_async(); // error C2338: co_await is not supported in coroutines
                      // of type std::experimental::generator
}
	SomeAsyncCoro<void> do_async();
	
	generator<int> g() {
	  yield(42);
	  do_async(); // UB
	}

All coroutines proposal except for Symmetric Coroutines provide an explicit lexical marker identifying a suspend point (co_yield, co_await and [<-]).

Symmetric Coroutines	

template <class R, class T>
R func(T u) {
  do1();
  u();
  do2();
}
	This template defines either a function “func” to a class func depending on what type R is during instantiation of a template.
	
	u() could be a suspend point or not, depending on the type T.

Note 1: The authors of Symmetric Coroutines are exploring alternatives to the coroutine definition syntax to avoid collisions with other language constructs and to support forward declarations..

Note 2: The authors of Symmetric Coroutines are open to a syntactic marker identifying a suspend point.

Coroutines TS today vs Core Coroutines / Symmetric Coroutines from the future

In this section we will consider use cases supported by Coroutines TS today and how it would look in other proposals when the issues preventing them from implementing asynchronous coroutines are fixed.

Actors

Actors programming model typically representing your program as a set of interacting objects sending messages to each other. Notationally, actors look like classes and message sends look like calls to member functions, however, processing of the message happens asynchronously on the dedicated strand of execution managed by the actor.

Coroutines TS via combination of parameter preview and coroutine_traits allows direct expression of actors in C++.

Coroutines TS

```
class MyActor : public Actor {
  task<int> request_reply(int a) {
    // executed asynchronously
    DoStuff(a);
    co_return 42;
  }
  oneway_task fire_and_forget() {
    // executed asynchronously, no
    // reply expected.
    co_await DoAsyncStuff();
    DoStuff(42);
  }
};
```

Symmetric Coroutines:

```
class MyActor : public Actor {
  Task<int> request_reply(int a) {
    // Demonstrates a problem with
    // the current syntax
    Task<int> request_reply_task(int a)
{
      // executed asynchronously
      DoStuff(a);
      return 42;
    }
    // Relies on a future customization
    // point
    return request_reply_task(a);
```

Core Coroutines

```
class MyActor : public Actor {
  // spawn is a base class method that posts
  // execution of the coroutine lambda on the
  // actor's strand/thread.

  auto request_reply(int a) {
    return spawn(
      [=]{ return
        [=][->] task<int> {
          // executed asynchronously
          DoStuff(a);
          return 42;
        });
    });
  }
  auto fire_and_forget() {
    return spawn(
      [=]{ return
        [=][->] oneway_task {
          // executed asynchronously, no
          // reply expected.
          [<-] DoAsyncStuff();
          DoStuff(42);
        });
      });
  }
};
```

```
  }

  void fire_and_forget() {
    OneWayTask workflow() {
      // executed asynchronously, no
      // reply expected.
      DoAsyncStuff();
      DoStuff(42);
    }

    OneWayTask::start<workflow>();
  }
};
```

Non-intrusive coroutinization

Core Coroutines proposal requires two intrusive change to an existing type to be able to author coroutines returning that type. 1. Add a nested shared_state_type class to any user defined class that maybe used as a return value of the coroutine. 2. Add a constructor to that type taking a coroutine state object as an argument. One of the design goals of Coroutines TS is incremental and seamless integration into existing codebases. C++ community is diverse. Some organizations have codebases where you can easily modify any line of the source code that goes into building your products, others, have code ownership restrictions that can make changes in libraries not owned by your team more challenging to make. Coroutines TS recognizes the diversity of C++ community and offers a non-intrusive way of introducing coroutines into the code base.

Coroutines TS

```
AsyncAction<int> f(int a) {
  ...
  co_return 42;
}
```

```
Symmetric Coroutines:

auto f(int a) {
 Task impl(int a) {
   …
   return 42;
 }
 return AsyncActionAdapter<impl>(a);
}
```

Core Coroutines

```
auto f(int a) {
  return AsyncActionAdapter(
      [=]{ return
        [=][->] task<int> {
          ...
          return 42;
        });
    });
  }
```

Controlling allocation for type-erased coroutines

If a user desires to control allocation of a type-erased coroutine it can do so without a lot of boilerplate under the Coroutines TS

Coroutines TS	Core Coroutines

`task<int> f(alloc_t, MyAlloc a) {`
`  DoStuff(a);`
`  co_return 42;`
`}`
	`auto request_reply(alloc_t, MyAlloc a) {`
	`    return alloc_on(a,`
	`      [=]{ return`
	`        [=][->] task<int> {`
	`          DoStuff(a);`
	`          return 42;`
	`        });`
	`    });`
	`  }`

Implicit Cancellation

Some asynchronous programming models include a convention where a cancellation_token, cancel_token is passed as an argument to a function with the expectation that it will be checked periodically and abort the computation if a cancellation was requested.
Coroutines TS supports implicitly augmenting every suspend point in the coroutine with a cancellation check via combination of parameter preview and await_transform features.

Coroutines TS	Core Coroutines

`task<> f(stop_token c) {`
`  …`
`  co_await g(c); // cancel point`
`  …`
`  co_await h(c); // cancel point`
`}`
	`auto f(stop_token c) [] [->] task<int> {`
	`  …`
	`  [<-] check_cancel(c, g(c));`
	`  …`
	`  [<-] check_cancel(c, h(c));`
	`}`

Note 1. While the code is not significantly larger in Core Coroutines, Coroutines TS eliminates the hazard of forgetting to check for cancellation before launching a potentially long async operation.

Merge and improve Coroutines TS possible shapes:

Future evolution of Coroutines TS that addresses the expected<T> and getting first-class state machine object can take several paths:

1) Unified coroutine approach
2) Coroutines TS + Core Coroutines
3) Manual/High-Sizeof solution for first class coroutines

4) Other

We would like to defer selection of the further evolution path until the feasibility of the alternatives have been explored sufficiently enough to understand the language and compiler impact of the options. So far, only option #3 was prototyped to verify implementability, but no usage experience was collected.

Additionally, once in the language and being used by large number of developers, the set of problems that committee considers as important to address in further revisions may be different than what it is now.

Conclusion

All proposals support mostly the same set of use cases with some strength and weaknesses shown earlier.

Core Coroutines out of the box offers the best story for **non-resuming** coroutines (such as unwrapping of expected) both in efficiency and simplicity of library bindings.
Coroutines TS provides the best near-term[3] efficiency for **asynchronous** type erased coroutines.
Core Coroutines is not planning to cover all use cases of Coroutines TS with the same convenience or near-term efficiency.

Coroutines TS incremental and Universal Coroutine Proposal expect to match the efficiency and usability of both Core Coroutines and Coroutines TS with the price being more user facing (both co_xxx keywords and [<-]) and more complicated library bindings than Coroutines TS or Core Coroutines by themselves.

Symmetric Coroutines differs from the other proposals in part because its lowest-level primitives are intended to be safe enough to be used directly by end users (in at least some scenarios), as well as efficient enough to be used to implement all coroutine library types. We do not yet have consensus as to whether it succeeds in this aim. It is also designed to provide interoperability among coroutine types "for free" (e.g. invoking an async coroutine from inside a generator), rather than via pairwise opt-in, but we do not have consensus as to whether this will provide safe and unsurprising behavior in practice.

[3] In the longer term (5+ years), equivalent performance gains may be achievable via non-coroutine-specific optimizer technology and language features.

References

N4775	Working Draft, C++ Extensions for Coroutines	Gor Nishanov
P0981R0	Halo: coroutine Heap Allocation eLision Optimization: the joint response	Richard Smith, Gor Nishanov
P1063R2	Core Coroutines	Geoff Romer, James Dennett, Chandler Carruth
P1342R0	Unifying Coroutines TS and Core Coroutines	Lewis Baker
P1362R0	Incremental Approach: Coroutine TS + Core Coroutines	Gor Nishanov
P1477R0	Coroutines TS Simplifications	Lewis Baker
P1430R0	First-class symmetric coroutines in C++	Mihail Mihaylov, Vassil Vassilev

Appendix

An example of how Incremental Coroutine TS + Core Coroutine integration might look like:

The following is how `expected<T>` unwrapping library binding might look in the incremental path. Note that coroutine_traits are not required. The only difference from Core Coroutines paper definition is an addition of "`get_return_object`" function.

```
struct promise_type {
  template <typename F>                                    // THIS IS NEW
  auto get_return_object(F f) { return f(*this); }   // THIS IS NEW

  template <template<typename> Continuation, typename U>
  expected<T, E> operator[<-](
    const expected<U,E>& e, Continuation<const T&>& continuation) {
    if (e.has_value()) {
       tail return continuation([&] { return *e; });
    } else {
      return unexpected(e.error());
    }
  }
  expected<T, E> operator return(const T& value) {
     return value;
  }
};
```

If compiler finds `get_return_object` that takes a single argument, it synthesizes a strongly typed coroutine object like that of Core Coroutines and gives it to `get_return_object`. The body of the `get_return_object` is free to wrap F in any way it wants. Otherwise, the language assumes Coroutines TS behavior and creates a type-erased coroutine exactly as it does today.