Document number: P1329R0

Date: 2018-11-2
Reply To:
Mihail Mihaylov (mmihailov@vmware.com)
Vassil Vassilev (v.g.vassilev@gmail.com)
Audience: WG21 Evolution Working Group

On the Coroutines TS

We have familiarized ourselves with the Coroutines TS [1] , and two other proposals - Core Coroutines [2] and Resumable Expressions [3]. We have also spent time experimenting with the Coroutine TS in a preexisting production code base. Based on this, our position is that the Coroutines TS should not be merged into the working draft. We have concerns about the design of the feature and in our opinion the high demand for the feature should not be a reason to adopt the design, if a better one is possible. Instead of adopting the Coroutines TS we would prefer improving the Core Coroutines proposal or exploring other alternatives.

Our concerns about the Coroutines TS fall into three categories – interface, terminology and performance.

Interface

Our most general concern is that coroutines as proposed in the Coroutines TS are not first class citizens of the C++ language. In the TS, a coroutine definition essentially defines a factory function which returns a type-erased handle for it. The actual coroutine has no type, and no scope. As such, coroutines cannot be used in many ways in which first class types can. They cannot be allocated with automatic lifetime, including being aggregated into other objects, they cannot be used for dispatch and in general are not as naturally composable with other elements of the language as first-class features.

We would favour a design which makes coroutines first class citizens of the language.

A specific concern is that the lifetime of the coroutine frame and the lifetime of the wrapper object that is created when the coroutine is instantiated are separate. As the promise object is allocated on the coroutine frame, this makes it harder for programmers to reason about the lifetime of values and exceptions returned by the coroutine.

We would favour a design where the coroutine frame is a part of the wrapper object (the coroutine lambda in the case of the Core Coroutines proposal), which would allow programmers to reason about the coroutine lifetime through the lifetime of the wrapper and offer experts better control of the coroutine frames.

Another issue is the policy-based customization. While this approach is very useful in many cases, in the context of the Coroutines TS over time it has evolved into a big number of customization points. We believe that it is possible to achieve the same level of customization by just overriding a small number of operations. Furthermore, the customization points in the Coroutines TS allow counterintuitive behavior. For example, it’s possible to define a promise where the co_yield expression will not yield.

We would favour a design which has a smaller and simpler customization interface.

Terminology

We echo the concern already expressed by others about the choice of the co_await keyword. The coroutine concept was introduced 60 years ago, and has well-established terminology. Co_await does not follow this terminology and focuses on a specific use case instead of on the essence of coroutines.

We share the concern regarding the risk of collisions between new keywords and existing identifiers used in user code. Still, the Coroutines TS proposal suggests adding three of them with the “co_” prefix. So we presume there is a general acceptance of new keywords starting with an uncommon prefix. In addition, other TSes seem to have reached consensus on stealing very popular identifiers.

We would favour a design which employs either the function call syntax or keywords that include in their spelling the verbs “yield”, “resume” and “suspend”, the way co_yield does.

Performance

We are concerned about the heap allocation of the coroutine frame. Our understanding is that the motivation for this design choice was to make it possible for suspended coroutines to leave the scope where they were created including “teleporting” them to other threads.

To address this concern, the Coroutines TS relies heavily on the heap elision optimization. We are concerned that there will be many use cases where the compiler won’t be able to determine correctly the lifetime of the coroutine frame, and will be forced to go with the more conservative estimate and keep the heap allocation.

In the context of coroutines we can expect that it will be hard for compilers to detect the optimization opportunity in cases when coroutines are aggregated into objects, nested within other coroutines, returned from factory functions, etc. The fact that the coroutine frame could easily outlive its wrapper object can further complicate the task of the compiler.

Following are some examples of cases when heap elision doesn't work (with the latest version of clang in Compiler Explorer at O2) [4].

First, let's define a simple coroutine wrapper class and two simple coroutines that use it:

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

using namespace std;

template<typename V> class RawCoroutine {
public:
   struct promise_type;
   using handle = std::experimental::coroutine_handle<promise_type>;

   struct promise_type {
      V value;
      exception_ptr e;

      auto get_return_object() {
         return RawCoroutine{handle::from_promise(*this)};
      }

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

      auto final_suspend() {
         return std::experimental::suspend_always();
      }

      auto return_value(V v) {
         value = v;
         return std::experimental::suspend_never();
      }

      void unhandled_exception() {
         e = current_exception();
      }
      auto yield_value(V v) {
         value = v;
         return std::experimental::suspend_always();
      }
   };

   RawCoroutine(const RawCoroutine&) = delete;

   RawCoroutine(RawCoroutine&& other) : _coro(other._coro) {
      other._coro = nullptr;
   }

   RawCoroutine(handle h) : _coro(h) {}

   ~RawCoroutine() {
      _coro.destroy();
   }

   V operator()() {
      _coro();
      if (_coro.promise().e)
         rethrow_exception(_coro.promise().e);
      else
         return _coro.promise().value;
   }

   bool done() {
      return _coro.done();
   };

private:
   handle _coro;
};

static RawCoroutine<int> SimpleRange() {
   for (int i = 0; i < 10; ++i)
      co_yield i;

   co_return 10;
}

static RawCoroutine<int> CompositeRange() {
   RawCoroutine<int> coro = SimpleRange();

   while (!coro.done())
      co_yield coro() * 2;
   co_return -1;
}

struct Base {
   virtual ~Base() {}
   virtual RawCoroutine<int> coro() = 0;
};

struct Derived : Base {
   RawCoroutine<int> coro() override {
      for (int i = 0; i < 10; ++i)
         co_yield i;

      co_return 10;
   }
};
```

Next, let's look at several examples where we use these coroutines and whether clang will detect that it can use heap elision.

For a simple scoped coroutine, clang applies heap elision:

```
int main() {
   int i = 0;

   RawCoroutine<int> coro = SimpleRange();

   while (!coro.done())
      i += coro();

   return i;
}
```

But it doesn't apply it if the coroutine wrapper itself is on the heap:

```
int main() {
   int i = 0;

   auto coro = make_unique<RawCoroutine<int>>(SimpleRange());
   while (!coro->done())
      i += (*coro)();

   return i;
}
```

Similarly, when the coroutine wrapper is itself wrapped in another object, clang elides the coroutine handle allocation when wrapper object is on the stack:

```
int main() {
   int i = 0;

   struct Wrapper {
      Wrapper() : coro(SimpleRange()) {}

      RawCoroutine<int> coro;
   };

   Wrapper wrapper;

   while (!wrapper.coro.done())
      i += wrapper.coro();

   return i;
}
```

But not when it's on the heap:

```
int main() {
   int i = 0;

   struct Wrapper {
      Wrapper() : coro(SimpleRange()) {}

      RawCoroutine<int> coro;
   };

   auto wrapper = make_unique<Wrapper>();
   while (!wrapper->coro.done())
      i += wrapper->coro();

   return i;
}
```

In the case of a nested coroutine, clang elides the heap allocation of the frame of the outer coroutine, but not of nested one:

```
int main() {
   int i = 0;

   RawCoroutine<int> coro = CompositeRange();

   while (!coro.done())
      i += coro();
   return i;
}
```

The compiler also fails to elide the allocation when the coroutine is virtual:

```
int main() {
   int i = 0;

   unique_ptr<Base> b = make_unique<Derived>();
   RawCoroutine<int> coro = b->coro();

   while (!coro.done())
      i += coro();

   return i;
}
```

The compiler elides the allocation when the coroutine is created by a factory function:

```
int main() {
   int i = 0;

   struct Factory {
      static RawCoroutine<int> Create() { return SimpleRange(); }
   };

   RawCoroutine<int> coro = Factory::Create();
   while (!coro.done())
      i += coro();

   return i;
}
```

But not when the factory function is in another translation unit (or not inlined for some other reason):

```
int main() {
   int i = 0;

   struct Factory {
      // No inline, to simulate another translation unit
      static RawCoroutine<int> Create() __attribute__((noinline)) {
         return SimpleRange();
      }
   };
```

```
    RawCoroutine<int> coro = Factory::Create();
    while (!coro.done())
       i += coro();

    return i;
 }
```

These are just some examples, but we can expect many more scenarios where the coroutine frame is allocated on the heap, even though the coroutine doesn't leave the scope where it's created.

More importantly, even if the heap elision optimization becomes completely reliable in the future, it will be hard for developers to predict when it will take place. This will make it hard for programmers to reason about the performance of their code.

For these reasons, we favour a design which makes it explicit how the coroutine frame is allocated.

Path forward

The Bulgarian NB is exploring alternative directions and we plan to present a preview of a proposal at the next meeting in Kona. Still, we feel that by opposing the Coroutines TS, we have the obligation to present at least a direction for improvement, so we are sharing a very preliminary view of what we are working on. We would favour a design:

- which makes coroutines first class citizens of the language;
- where the coroutine frame is a part of the wrapper object (the coroutine lambda in the case of the Core Coroutines proposal), which would allow programmers to reason about the coroutine lifetime through the lifetime of the wrapper and offer experts better control of the coroutine frames;
- which has a smaller and simpler customization interface;
- which employs either the function call syntax or keywords that include in their spelling the verbs "yield", "resume" and "suspend", the way co_yield does;
- which makes it explicit how the coroutine frame is allocated.

The core idea in our future proposal is for coroutine definitions to define classes instead of factory functions. The pseudo code:

```
 std::coroutine<int> Range(int start, int end) {
    for (int i = start; i < end - 1; ++i)
       yield(i);
    return end;
}
```

Would declare a class Range that derives from std::coroutine<int> which has in its memory layout reserved space for the captures and the coroutine frame and suspension point:

```
class Range : public std::coroutine<int> {
   coroutine_state<Range> frame; // Not visible in
                                 // the coroutine body

   int start; // Visible in the coroutine body
   int end; // Visible in the coroutine body

public:
   Range(int start, int end)
      : coroutine<int>(...)
      , start(start)
      , end(end)
   {}

   int operator()() {
      // Transformed function body
      // ...
   }
}
```

The yield(i) statement would be a method of the base class std::coroutine<int>, so it would be only available in the context of a coroutine and would have a low risk of clashing with user identifiers. If that is not acceptable, we would opt for keywords like co_yield, co_suspend and co_return.

Once defined, the coroutine can be used as any functor object that the programmer could write by hand:

```
int main() {
   // Automatic storage duration.
   Range rangeScoped(0, 10);

   int i = 0;
   while (!rangeScoped.done())
      i += rangeScoped();

   // Dynamic storage duration.
   auto rangeDyn = make_unique<Range>(0, 10);

   while (!ran10geDyn->done())
      i += (*rangeDyn)();
```

```
    // Passing by reference
    Range r(10, 20);
    i += foo(r);

    // Aggregation in other classes
    struct Wrapper {
       Wrapper(int end) : range(0, end) {}

       Range range;
    };

    return i;
}
```

The proposed lowering also allows programmers to create their own coroutine types by deriving from the base `std::coroutine` type.

This change or a similar one can be applied directly to the Coroutines TS design as well as to the Core Coroutines proposal. But if accepted, such a change to any of these proposals will necessarily delay its merging at least until the meeting in Kona. So we would like to take this time to develop our proposal further, as we are interested in exploring other sides of the coroutines design too. We intend to work closely with Gor Nishanov and the authors of the Core Coroutines proposal and we will strongly consider changes to the existing proposals instead of a completely new one, if possible.

Acknowledgements

The authors would like to thank Anton Stoyanov, Aleksandar Cheshmedjiev, Georgi L Dimitrov, Hristosko Chaushev, Petko Padevski, Peter Dimov, Stanimir Lukanov and Viktor Kaltchev for comments on early drafts of this proposal.

Thanks again to all those involved in the Coroutines TS, and particularly Gor Nishanov, for exploring the coroutines design space and providing a basis for this feedback paper.

References

[1] Coroutines TS, N4760, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4760.pdf
[2] Core Coroutines, P1063R1, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1063r1.pdf
[3] Resumable expressions, P0114R0, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0114r0.pdf
[4] https://godbolt.org/z/y8nB_u