`std::generator`: Synchronous Coroutine Generator for Ranges

Document #: P2168R0
Date: 2020-05-16
Project: Programming Language C++
Audience: LEWG

Reply-to: Lewis Baker <lbaker@fb.com >
Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose a standard library type `std::generator` which implements a coroutine generator compatible with ranges.

Example

```
std::generator<int> fib (int max) {
    co_yield 0;
    auto a = 0, b = 1;

    for(auto n : std::views::iota(0, max))  {
        auto next = a + b;
        a = b, b = next;
        co_yield next;
    }
}

int answer_to_the_universe() {
    auto coro = fib(7) ;
    return std::accumulate(coro | std::views::drop(5), 0);
}
```

Motivation

C++ 20 had very minimalist library support for coroutines. Synchronous generators are an important use case for coroutines, one that cannot be supported without the machinery presented in this paper. Writing an efficient and correctly behaving recursive generator is non-trivial, the standard should provide one.

Design

While the proposed `std::generator` interface is fairly straight-forward, a few decisions are worth pointing out.

`input_view`

`std::generator` is a non-copyable `view` which models `input_range` and spawn move-only iterators. This is because the coroutine frame is a unique resource (even if the coroutine *handle* is copyable). Unfortunately, some generators can satisfy the `view` constraints but fail to model the `view` O(1) destruction requirement:

```
template <typename T>
std::generator<T> all (vector<T> vec) {
    for(auto & e : vec)  {
        co_yield e;
    }
}
```

Header

Multiple options are available as to where put the `generator` class.

- `<coroutine>`, but `<coroutine>` is a low level header, and `generator` depends on bits of `<type_traits>` and `<iterator>`.
- `<ranges>`
- A new `<generator>`

Separately specifyable Value Type

This proposal supports specifying both the ”yielded” type, which is the iterator ””reference”” type (not required to be a reference) and its corresponding value type. This allow ranges to handle proxy types and wrapped `reference`, like this implementation of `zip`:

```
template<std::ranges::input_range Rng1,
         std::ranges::input_range Rng2>
generator<
    std::tuple<std::ranges::range_reference_t<Rng1>,
               std::ranges::range_reference_t<Rng2>,
    std::tuple<std::ranges::range_value_type_t<Rng1>,
               std::ranges::range_value_type_t<Rng2>>>
zip(Rng1 r1, Rng2 r2) {
    auto it1 = std::ranges::begin(r1);
    auto it2 = std::ranges::begin(r2);
    auto end1 = std::ranges::end(r1);
    auto end2 = std::ranges::end(r2);
    while (it1 != end1 && it2 != end2) {
        co_yield {*it1, *it2};
        ++it1; ++it2;
    }
}
```

Recursive generator

A ”recursive generator” is a coroutine that supports the ability to directly `co_yield` a generator of the same type as a way of emitting the elements of that `generator` as elements of the current `generator`.

Example: A `generator` can `co_yield` other generators of the same type

```
generator<const std::string&> delete_rows(std::string table, std::vector<int> ids) {
  for (int id : ids) {
    co_yield std::format("DELETE FROM {0} WHERE id = {1}", table, id);
  }
}

generator<const std::string&> all_queries() {
  co_yield delete_rows("user", {4, 7, 9 10});
  co_yield delete_rows("order", {11, 19});
}
```

Example: A `generator` can also be used recursively

```
struct Tree {
  Tree* left;
  Tree* right;
  int value;
};

generator<int> visit(Tree& tree) {
  if (tree.left) co_yield visit(*tree.left);
  co_yield tree.value;
  if (tree.right) co_yield visit(*tree.right);
}
```

In addition to being more concise, the ability to directly yield a nested generator has some performance benefits compared to iterating over the contents of the nested generator and manually yielding each of its elements.

Yielding a nested `generator` allows the consumer of the top-level coroutine to directly resume the current leaf generator when incrementing the iterator, whereas a solution that has each generator manually iterating over elements of the child generator requires O(depth) coroutine resumptions/suspensions per element of the sequence.

Example: Non-recursive form incurs O(depth) resumptions/suspensions per element and is more cumbersome to write

```
generator<int> slow_visit(Tree& tree) {
  if (tree.left) {
    for (int x : visit(*tree.left))
      co_yield x;
  }
  co_yield tree.value;
  if (tree.right) {
    for (int x : visit(*tree.right))
      co_yield x;
  }
}
```

Exceptions that propagate out of the body of nested `generator` coroutines are rethrown into the parent coroutine from the `co_yield` expression rather than propagating out of the top-level ‘iterator::operator++()‘. This follows the mental model that ‘co_yield someGenerator‘ is semantically equivalent to manually iterating over the elements and yielding each element.

For example: `nested_ints()` is semantically equivalent to `manual_ints()`

```
generator<int> might_throw() {
  co_yield 0;
  throw some_error{};
}

generator<int> nested_ints() {
  try {
    co_yield might_throw();
  } catch (const some_error&) {}
  co_yield 1;
}

// nested_ints() is semantically equivalent to the following:
generator<int> manual_ints() {
  try {
    for (int x : might_throw()) {
      co_yield x;
    }
  } catch (const some_error&) {}
  co_yield 1;
}

void consumer() {
  for (int x : nested_ints()) {
    std::cout << x << " "; // outputs 0 1
  }

  for (int x : manual_ints()) {
    std::cout << x << " "; // also outputs 0 1
  }
}
```

The recursive form can be implemented efficiently with symmetric transfer. Earlier works in [CppCoro] implemented this feature in a distinct `recursive_generator` type.

However, it appears that a single type is reasonably efficient thanks to HALO optimizations and symmetric transfer. The memory cost of that feature is 3 extra pointers per generator. It is difficult to evaluate the runtime cost of our design given the current coroutine support in compilers. However our tests show no noticeable difference between a `generator` and a `recursive_generator` which is called non recursively. It is worth noting that the proposed design makes sure that HALO [3] optimizations are possible.

While we think a single `generator` type is sufficient and offers a better API, there are three options:

- A single `generator` type supporting recursive calls (this proposal).
- A separate type `recursive_generator` that can yield values from either `recursive_generator` or a `generator`. That may offer very negligible performance benefits, same memory usage.

  A separate recursive_generator type which can only yield values from other `recursive_generator`.

  That third option would make the following ill-formed:

  ```
  generator<int> f();
  recursive_generator<int> g() {
      co_yield f(); // incompatible types
  }
  ```

  Instead you would need to write:

  ```
  recursive_generator<int> g() {
    for (int x : f()) co_yield x;
  }
  ```

  Such a limitation can make it difficult to decide at the time of writing a generator coroutine whether or not you should return a `generator` or `recursive_generator` as you may not know at the time whether or not this particular generator will be used within `recursive_generator` or not.

  If you choose the `generator` return-type and then later someone wants to yield its elements from a `recursive_generator` then you either need to manually yield its elements one-by-one or use a helper function that adapts the `generator` into a `recursive_generator`. Both of these options can add runtime cost compared to the case where the generator was originally written to return a `recursive_generator`, as it requires two coroutine resumptions per element instead of a single coroutine resumption.

  Because of these limitations, we are not recommending this approach.

How to store the yielded value in the promise type?

The yielded expression is guaranteed to be alive until the coroutine resumes, it is, therefore, sufficient to store its address. This makes `generator` with a large yielded type efficient. However, it might pessimize yielding values smaller than a pointer because of the added indirection. (It is unclear what the cost of this indirection is, as none of these accesses should result in cache misses).

More annoyingly, this prevents conversions in yielding expressions:

```
generator<string_view> f() {
    co_yield std::string(); // error: cannot convert std::string to std::string_view &
}
```

Storing a copy would allow less indirection and the ability to yield any values convertible to the yielded type, at the cost of more storage. To avoid that storage cost, a `generator<const T&>` can be used.

Given at the value has to be stored in the coroutine frame anyway, it might interesting to add a `yield_transform` customization point to the core language, to either prevent the value to be stored in the coroutine frame or to transform its type, both would allow supporting yielding convertible type at no extra memory cost.

Future Work

A non-throwing default allocator with a noexcept generator function should permit not to force the cost of exceptions on users of this type.

Extend the ability to `co_yield` another generator of the same type to instead allow a generator to `co_yield` an arbitrary `range` with compatible element types. More investigation is required to resolve potential ambiguities when yielding types that are both a `range` and that are convertible to the `generator::reference` type.

Implementation and experience

`generator` has been provided as part of cppcoro and folly. However, cppcoro offers a separate `recursive_generator` type, which is different than the proposed design.

Folly uses a single `generator` type which can be recursive but doesn't implement symmetric transfer. Despite that, Folly users found the use of `Folly:::Generator` to be a lot more efficient than the eager algorithm they replaced with it.

`ranges-v3` also implements a `generator` type, which is never recursive and predates the work on move-only views and iterators [1], [2] which forces this implementation to ref-count the coroutine handler.

Our implementation [Implementation] consists of a single type which takes advantage of symmetric transfer to implement recursion efficiently.

Wording

The following wording is meant to illustrate the proposed API.

� Header `<coroutine>` synopsis [coroutine.syn]

[...]

```
namespace std {

template<typename Y, typename V  = std::remove_cvref_t<Y>>
class generator;

template <typename Y, typename V>
inline constexpr bool ranges::enable_view<generator<Y, V>> = true;


}
```

� Generator View [coroutine.generator]

� Overview [coroutine.generator.overview]

`generator` produces an `input_view` over a synchronous coroutine function yielding values.

[*Example:*

```
generator<int> iota(int start = 0) {
    while(true)
        co_yield start++;
}

void f() {
    for(auto i : iota() | views::take(3))
        cout << i << " " ; // prints 0 1 2
}
```

—*end example*]

� Class template `generator` [coroutine.generator.class]

```
namespace std {

template <typename Y, typename V = std::remove_cvref_t<Y>>
class generator  {
public:
    class promise_type;
    class iterator;
    class sentinel {};

private:
    std::coroutine_handle<promise_type> coroutine_ = nullptr; // exposition only
    explicit generator(std::coroutine_handle<promise_type> coroutine) noexcept // exposition
only
    : coroutine_(coroutine) {}

public:
    generator() noexcept;
    generator(const generator &other) = delete;
    generator(generator && other) noexcept
        : coroutine_(exchange(other.coroutine_, nullptr)){}

    ~generator() {
       if (coroutine_) {
           coroutine_.destroy();
       }
    }

    generator &operator=(generator && other) noexcept {
        swap(other);
        return *this;
    }

    iterator begin();
    sentinel end() noexcept
    { return {};  }

    void swap(generator & other) noexcept {
        std::swap(coroutine_, other.coroutine_);
    }

};
```

```
iterator begin();
```

Preconditions: `!coroutine_` is `true` or `coroutine_` refers to a coroutine suspended at its initial suspend-point.

Effects: Equivalent to:

```
if(coroutine_)
    coroutine_.resume();
return iterator{coroutine_};
```

[*Note:* It is undefined behavior to call `begin` multiple times on the same coroutine. —*end note*]

� Class template `generator::promise_type` [coroutine.generator.promise]

```
template <typename Y, typename V>
class generator<Y, V>::promise_type {
    friend generator;

public:
    using value_type = V;
    using reference  = Y;

    generator<Y, V> get_return_object() noexcept;

    std::suspend_always initial_suspend() const {
        return {};
    }
    auto final_suspend() const;

    std::suspend_always
    yield_value(reference && value) noexcept;

    \unspec yield_value(generator<Y, V>&& g) noexcept; // see below

    reference value() const; // exposition only

    void await_transform() = delete;

    void return_void() noexcept {}

    void unhandled_exception();
};
```

```
generator<Y, V> get_return_object() noexcept;
```

Effects: Equivalent to:

```
return generator<Y, V>{
    std::coroutine_handle<promise_type>::from_promise(*this)};
```

```
std::suspend_always
yield_value(reference && value) noexcept;
```

Effects: Store a reference to `value` in the promise.

```
auto yield_value(generator&& g) noexcept;
```

Effects: This function returns an implementation defined `awaitable` type which takes ownership of the generator `g`.

[*Note:* This ensures that local variables in-scope in `g`'s coroutine are destructed before local variables in-scope in this coroutine being destructed. — *end note*]

Execution is transferred to the coroutine represented by `g.coroutine_` until its completion. After `g.coroutine_` completes, the current coroutine is resumed.

[*Note:* Generators can transfer control recursively. — *end note*]

Effects: Returns the value previously set by a call to `yield_value`.

If the execution control has been transferred from this promise to another `generator`, `value` returns the value set on the promise associated with that generator instead.

[*Note:* Generators can transfer control recursively, `value` returns the value set on promise associated to the child-most generator coroutine. — *end note*]

� Class template `generator::iterator` [coroutine.generator.iterator]

```
template <typename Y, typename V>
class generator::iterator {
private:
    std::coroutine_handle<promise_type> coroutine_ = nullptr;


public:
    using iterator_category = std::input_iterator_tag;
    using difference_type = std::ptrdiff_t;
    using value_type = promise_type::value_type;
    using reference = promise_type::reference;

    iterator() noexcept = default;
    iterator(const iterator &) = delete;


    iterator(iterator && other) noexcept
    : coroutine_(exchange(other.coroutine_, nullptr)) {}

    iterator &operator=(iterator &&other) noexcept {
        coroutine_ = exchange(other.coroutine_, nullptr);
    }

    explicit iterator(std::coroutine_handle<promise_type> coroutine) noexcept
    : coroutine_(coroutine) {}

    bool operator==(sentinel) const noexcept {
         return !coroutine_ || coroutine_.done();
    }

    iterator &operator++();
    void operator++(int);

    reference operator*() const noexcept;
    reference operator->() const noexcept requires std::is_reference_v<reference>;

};
```

```
iterator &operator++();
```

Preconditions: `coroutine_ && !coroutine_.done()` is `true`.

Effects: Equivalent to:

```
coroutine_.resume();
return *this;
```

```
void operator++(int);
```

Preconditions: `coroutine_ && !coroutine_.done()` is `true`.

Effects: Equivalent to:

```
(void)operator++();
```

```
reference operator*() const noexcept;
reference operator->() const noexcept requires std::is_reference_v<reference>;
```

Preconditions: `coroutine_ && !coroutine_.done()` is `true`.

Effects: Equivalent to:

```
return coroutine_.promise().value();
```

References

[1] Casey Carter. P1456R1: Move-only views. https://wg21.link/p1456r1, 11 2019.

[2] Corentin Jabot. P1207R0: Movability of single-pass iterators. https://wg21.link/p1207r0, 8 2018.

[3] Richard Smith and Gor Nishanov. P0981R0: Halo: coroutine heap allocation elision optimization: the joint response. https://wg21.link/p0981r0, 3 2018.

[CppCoro] Lewis Baker *CppCoro: A library of C++ coroutine abstractions for the coroutines TS*
https://github.com/lewissbaker/cppcoro

[Folly] Facebook *Folly: An open-source C++ library developed and used at Facebook*
https://github.com/facebook/folly

[range] Eric Niebler *range-v3 Range library for C++14/17/20*
https://github.com/ericniebler/range-v3

[Implementation] Lewis Baker, Corentin Jabot *`std::generator` implementation*
https://godbolt.org/z/icfqLr

[N4861] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4861