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

Document #:	P2168R2
Date:	2021-03-22
Project:	Programming Language C++
Audience:	LEWG

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

Abstract

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

Revisions

R2

- Some wording fixes
- Improve the section on allocator support
- Updated implementation

R1

- Add benchmarks results and discussion about performance
- Introduce `elements_of` to avoid ambiguities when a generator is convertible to the reference type of the parent generator.
- Add allocator support
- Symmetric transfer works with generators of different value / allocator types
- Remove `iterator::operator->`
- Put `generator` in a new `<generator>` header.
- Add an other example to motivate the `Value` template parameter

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 state 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 <class 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;
    }
}
```

In this second example, using string as value type ensures that calling code can take the necessay steps to make sure iterating over a generator would not invalidate any of the yielded values

```
// Yielding string literals : always fine
std::generator<std::string_view> string_views() {
    co_yield "foo";
    co_yield "bar";
}

std::generator<std::string_view, std::string> strings() {
    co_yield "start";
    std::string s;
    for (auto sv : string_views()) {
        s = sv;
        s.push_back('!');
        co_yield s;
    }
    co_yield "end";
}
```

```
// conversion to a vector of strings
// If the value_type was string_view, it would convert to a vector of string_view,
// which would lead to undefined beavior as the string_views may get invalidated upon iteration!
auto v = std::ranges::to<vector>(strings()); // (P1206R3 [3])
```

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 elements_of(delete_rows("user", {4, 7, 9 10}));
    co_yield elements_of(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 elements_of(visit(*tree.left));
    co_yield tree.value;
    if (tree.right) co_yield elements_of(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 : elements_of(visit(*tree.left)))
        co_yield x;
    }
    co_yield tree.value;
    if (tree.right) {
        for (int x : elements_of(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 elements_of(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
    }
}
```

`elements_of`

`elements_of` is a utility function that prevents ambiguity when a nested generator type is convertible to the value type of the present generator

```
generator<int> f()
{
    co_yield 42;
}

generator<any> g()
{
    co_yield f(); // should we yield 42 or generator<int> ?
}
```

To avoid this issue, we propose that:

- `co_yield <expression>` always yield the value directly.
- `co_yield elements_of(<expression>)` yield the values of the nested generator.

For convenience, we further propose that `co_yield elements_of(x)` be extended to support yielding the values of arbitrary ranges beyond generators, ie

```
generator<int> f()
{
    std::vector<int> v = /*... */;
    co_yield elements_of(v);
}
```

Symmetric transfer

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 [5] 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.

Symmetric transfer is possible for different generator types as long as the `reference` type is the same, aka, different value type or allocator type does not preclude symmetric transfer (see sectiion on allocators).

How to store the yielded value in the promise type?

When the `reference` type of `generator` is not a reference, A copy of the yielded value need to be stored, to support both rvalue references and yielding values of different types (which are convertible_to `reference`).

There are multiple implementation strategies possible to store the value in the generator. A previous revision of this paper always stored a copy of the yielded value. When the reference type was not a reference, this led to two copies: - One at the point of the co_yield expression - One when calling the `iterator::operator*`.

However, the yielded expression is guaranteed to be alive until the coroutine resumes, it is, therefore, sufficient to store its address.

We can take advantage of that fact by only storing a pointer in the generator. When a copy needs to be made by `yield_value` (because the yielded value is not of the same type, or cannot bind to the reference type), we can store the value in an awaiter that will remain alive until the end of the co_yield expression.

```
std::suspend_always yield_value(const Ref& x) {
    auto &root = rootOrLeaf_.promise();
    root.valuePtr_ = std::addressof(x);
    return {};
}

std::suspend_always yield_value(Ref& x) {
    auto &root = rootOrLeaf_.promise();
    root.valuePtr_ = std::addressof(x);
    return {};
}

template<typename T = std::remove_cvref_t<Ref>>
requires !is_reference_v<Ref> && is_constructible_v<Ref, T>
auto yield_value(T&& x) {
    struct yield_value_holder {
        Ref ref;

        bool await_ready() noexcept { return false; }

        template<typename Promise>
        void await_suspend(coroutine_handle<Promise> h) noexcept {
            h.promise().valuePtr_ = addressof(ref);
        }
        void await_resume() noexcept {}
    };
    return yield_value_holder{forward<T>(x)};
}
```

A drawback of this solution is that the yielded value is only destroyed at the end of the full expression:

```
(co_yield x, co_yield y); // x is destroyed after y is yielded.
```

We think this is a reasonable tradeof as it avoids a copy. Further optimization could be done to copy small values - and avoid an indirection. but it is unclear what the cost of this indirection is, as none of these accesses should result in cache misses).

To support different implementation strategies to store the value, the wording does not specify a type for the return value of `yield_value`.

Allocator support

In line with the design exploration done in section 2 of P1681R0 [4], `std::generator` can support both stateless and stateful allocators, and strive to minimize the interface verbosity for stateless allocators, by templating both the generator itself and the `promise_type`'s `new` operator on the allocator type. Details for this interface are found in P1681R0 [4].

`coroutine_parameter_preview_t` such as discussed in section 3 of P1681R0 [4] has not been explored in this paper.

```
std::generator<int> stateless_example() {
    co_yield 42;
}

template <class Allocator>
std::generator<int>
allocator_example(std::allocator_arg_t, Allocator alloc) {
    co_yield 42;
}

my_allocator<std::byte> alloc;
input_range auto rng = allocator_example<my_allocator<std::byte>>(std::allocator_arg, alloc);
```

The proposed interface requires that, if an allocator is provided, it is the second argument to the coroutine function, immediately preceded by an instance of `std::allocator_arg_t`. This approach is necessary to distinguish the allocator desired to allocate the coroutine state from allocators whose purpose is to be used in the body of the coroutine function. The required argument order might be a limitation if any other argument is required to be the first, however, we cannot think of any scenario where that would be the case.

We think it is important that all standard and user coroutines types can accommodate similar interfaces for allocator support. In fact, the implementation for that allocator support can be shared amongst `generator`, `lazy` and other standard types.

By default `std::generator` type erases the allocator type, and uses `std::allocator` unless an allocator is provided to the coroutine function. Then:

Type erased allocator(default)

```
template <class Allocator>
std::generator<int> f(std::allocator_arg_t, Allocator alloc) {}

f(std::allocator_arg, my_alloc{});
```

Returns a generator of type `std::generator<int, int, void>` where `void` denotes that the allocator is type erased. The allocator is store on the coroutine state if it is stateful or not default constructible; a pointer is always stored so that the `deallocate` method of the type erased allocator can be called.

No allocator

```
std::generator<int> f() {}
f();
```

Returns a generator of type `std::generator<int, int, void>` where `void` denotes that the allocator is type erased. A pointer is stored so that the `deallocate` method of the type erased allocator can be called. The allocator is `std::allocator` and is not stored on the frame (because it is stateless)

Explicit stateless allocator

```
std::generator<int, int, std::stateless_allocator<int>> f() {}
f();
```

Returns a generator of type `std::generator<int, int, std::stateless_allocator<int>>` No extra storage is used for the allocator because it is stateless.

Explicit stateful allocator

```
std::generator<int, int, some_statefull_allocator<int>>
    f(std::allocator_arg_t,  some_statefull_allocator<int> alloc) {}
f(std::allocator_arg, some_allocator); // must be convertible to some_statefull_allocator
```

Returns a generator of type `std::generator<int, int, some_statefull_allocator<int>>` The allocator is copied in the coroutine state.

Interaction with symmetric transfer and allocator support

The allocator must be part of the promise type. Or implementation uses a base class so that generators of different allocator types can yield each other. This leaves with 2 implementation strategies

- Storing a pointer to the base class in the promise handle
- Leave implementers find the best implementation strategy and can bell their own implementation to be well-formed
- Modify the wording as follow to allow `coroutine_handle<promise_base>::from_address(coro.address())` to be well-formed:

To support yielding nested generator of different allocator types we then have several options:

- Leaving implementers find the best strategy with compiler magic
- Modify from_address to allow construcruction from a coroutine of layout compatible type

  ```
  static constexpr coroutine_handle<Promise> coroutine_handle<Promise>::from_address(void* addr);
  ```

  > *Expects:* `addr` was obtained via a prior call to `address` on an object of type *cv*~~`coroutine_handle<Promise>`~~ of type *cv*`coroutine_handle<T>` where`T` is any type such that `is_layout_compatible_v<T, Promise>` is `true`.

Supporting allocators requires storing, in all cases, a function pointer adjacent to the coroutine state (to track a deallocation function), along with the allocator itself in the case of stateful allocators.

Can we postpone adding support for allocator later?

A case can be made that allocator support could be added to `std::generator` later. However, because the proposed design has the allocator as a template parameter, adding allocator after `std::generator` ships would represent an ABI break. We recommend that we add allocator support as proposed in this paper now and make sure that the design remains consistent as work on `std::lazy` is made in this cycle. However, it would be possible to extend support for different mechanisms (such as presented in section 3 of P1681R0 [4] later.

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 that takes advantage of symmetric transfer to implement recursion.

Performance & benchmarks

Because implementations are still being perfected, and because performance is extremely dependant on whether HALO optimization (see P0981R1 [**?**]) occurs, it is difficult at this time to make definitive statements about the performance of the proposed design.

At the time of the writing of this paper, Clang is able to inline non-nested coroutines whether the implementation supports nested coroutines or not, while GCC never performs HALO optimization.

When the coroutine is not inlined, support for recursion does not noticeably impact performance. And, when the coroutine yields another generator, the performance of the recursive version is noticeably faster than yielding each element of the range. This is especially noticeable with deep recursion.

	Clang	Clang ST[1]	GCC	GCC ST[1]	MSVC	MSVC ST[1]
Single value	(1) 0.235	(2) 2.36	12.4	13.4	61.9	63.7
Single value, noinline (3)	13.5	13.7	14.1	15.2	63.8	64.4
Deep nesting	43670266.0	(4) 427955.0	58801348	338736	224052033	4760914

[1] Symmetric transfer.

The values are expressed in nanoseconds. However, please note that the comparison of the same result across compiler is not meaningful, notably because the MSVC results were obtained on different hardware. That being said we observe:

- Only Clang can perform constant folding of values yielded by simple coroutine (1)
- When the `generator` supports symmetric transfer, clang is not able to fully inline the generator construction, but HALO is still performed (2).
- When HALO is not performed, the relative performance of both approach is similar (3).
- Supporting recursion is greatly beneficial to nested/recursive algorithms (4).

The code for these benchmarks as well as more detailled results can be found on Github.

Wording

The following wording is meant to illustrate the proposed API.

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

```
namespace std::ranges {

...

template<input_or_output_iterator I, sentinel_for<I> S, subrange_kind K>
inline constexpr bool enable_borrowed_range<subrange<I, S, K>> = true;

// ??, dangling iterator handling
struct dangling;

template<std::ranges::input_range R>
struct elements_of;

template<range R>
using borrowed_iterator_t = conditional_t<borrowed_range<R>, iterator_t<R>, dangling>;

...

}
```

� ranges::element_of [ranges.elementsof]

`elements_of` is a type that encapsulates a range and acts as a tag in overload sets to disambiguate when a range should be treated as a sequence rather than a single value in genetric contexts.

[*Example:*

```
generator<any> f(input_range auto rng) {
    co_yield rng; // yield rng as a single value
```

```
            co_yield elements_of(rng); // yield each element of rng
        }
```

— end example]

```
namespace std::ranges {
    template<std::ranges::input_range R>
    struct elements_of {
        R&& range_; // exposition only

        explicit constexpr elements_of(R&& r) noexcept;

        constexpr elements_of(elements_of&&) = default;

        elements_of(const elements_of&) = delete;
        elements_of& operator=(const elements_of&) = delete;
        elements_of& operator=(elements_of&&) = delete;

        constexpr R && get() && noexcept;
    };
    template<std::ranges::input_range R>
    elements_of(R&& r) -> return elements_of<R>;

}
```

`explicit constexpr elements_of(R&& r) noexcept;`

Effects: Initializes `range_` with `forward<R>(r)`.

`constexpr R && get() && noexcept;`

Returns: `forward<R>(range_)`.

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

```
#include <coroutine>
#include <ranges>

namespace std {

    template<class Ref,
        common_reference_with<Ref> Value = remove_cvref_t<Ref>,
        class Allocator = void>
    class generator;

    template <class Ref, class Value, class Allocator>
    inline constexpr bool ranges::enable_view<generator<Ref, Value, Allocator>> = true;
}
```

� Generator View [coroutine.generator]

� Overview [coroutine.generator.overview]

`generator` generates a sequence of elements by repeatedly resuming the coroutine it was returned from. When the coroutine is resumed, it is executed until it reaches either a `co_yield` expression or the end of the coroutine.

Elements of the sequence are produced by the coroutine each time a `co_yield` expression is evaluated.

When the `co_yield` expression is of the form `co_yield elements_of(rng)`, each element of the range `rng` is successively produced as an element of the generator.

`generator` models `view` and `input_view`.

[*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 <class Ref, common_reference_with<Ref> Value = remove_cvref_t<Ref>,
          class Allocator = void>
class generator  {
    class promise_type_; // exposition only

public:
    using promise_type = promise_type_;

    class iterator; // exposition only

    generator() noexcept = default;
    generator(const generator &other) = delete;
    generator(generator && other) noexcept;

    ~generator();

    generator &operator=(generator other) noexcept;

    iterator begin();
    sentinel end() const noexcept;

private:
    std::coroutine_handle<promise_type> coroutine_ = nullptr; // exposition only
    bool started_ = false; // exposition only

    explicit generator(std::coroutine_handle<promise_type> coroutine) noexcept; // exposition only

};
}
```

- `coroutine_traits<generator<Ref, Value, Allocator>>::promise_type` is valid an denotes a type,
- `Allocator` either meets the `Cpp17Allocator` requirements or is `void`.

`generator(std::coroutine_handle<promise_type> coro) noexcept;`

Initializes `coroutine_` with `coro`.

`generator(generator &&other) noexcept;`

Initializes `coroutine_` with `exchange(other.coroutine_, {})`,
and `started_` with `exchange(other.started_, false)`.

`~generator() noexcept;`

Effects: equivalent to:

```
if (coroutine_) {
    if (started_ && !coroutine_.done()) {
```

```
                coroutine_.promise().destruct_value_();
            }
            coroutine_.destroy();
        }
```

```
generator &operator=(generator other) noexcept;
```

Effects: equivalent to:

```
swap(coro_, other.coro_);
swap(started_, other.started_);
```

```
iterator begin();
```

Preconditions:

- `!coroutine_` is `true` or `coroutine_` refers to a coroutine suspended at its initial suspend-point,
- `started_` is false.

Effects: Equivalent to:

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

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

```
default_sentinel_t end() const noexcept;
```

Returns: `default_sentinel_t{}`.

� Exposition-only class template `generator::promise_type_` [coroutine.generator.promise]

```
template <class Ref, class Value, class Allocator>
class generator<Ref, Value, Allocator>::promise_type_ {

    friend generator;

    union {
        Ref value_; // exposition only
    };

    void destruct_value_() { // exposition only
        if constexpr(!is_lvalue_reference_v<Ref>) {
            value_.~decay_t<Ref>();
    }

public:

    generator<Ref, Value, Allocator> get_return_object() noexcept;

    suspend_always initial_suspend() noexcept;

    auto final_suspend() noexcept;

    unspecified yield_value(const Ref & value)
    noexcept(is_nothrow_move_constructible_v<Ref>);

    template <class T>
    requires is_convertible_v<T, Ref>
    unspecified yield_value(T&& x) noexcept(is_nothrow_constructible_v<Ref, T>);

    template <class TVal, class TAlloc>
    unspecified yield_value(elements_of<generator<Ref, TVal, TAlloc>> g) noexcept; // see below

    template<ranges::input_range R>
    requires convertible_to<ranges::range_reference_t<R>, Ref>
    unspecified yield_value(elements_of<R> rng); // see below

    void await_transform() = delete;

    void return_void() noexcept {};

    void unhandled_exception();

    static void* operator new(size_t size) requires same_as<Allocator, void>;
    static void* operator new(size_t size) requires (!same_as<Allocator, void> && is_default_constructible_v<

    template<typeame Alloc, class... Args>
    static void* operator new(size_t size, allocator_arg_t, Alloc& alloc, Args&...);

    template<class This, typeame Alloc, class... Args>
    static void* operator new(size_t size, This&, allocator_arg_t, Alloc& alloc, Args&...);

    static void operator delete(void* pointer, size_t size) noexcept;
};
```

```
generator<Ref, Value, Allocator> get_return_object() noexcept;
```

Effects: Equivalent to:

```
return generator<Ref, Value, Allocator>{
    coroutine_handle<promise_type>::from_promise(*this)};
```

```
suspend_always initial_suspend() noexcept;
```

Returns: `suspend_always{}`.

```
unspecified yield_value(const Ref& x)
    noexcept(is_nothrow_move_constructible_v<Ref>);

template <class T>
requires is_convertible_v<T, Ref>
unspecified yield_value(T&& x)
    noexcept(is_nothrow_constructible_v<Ref, T>);
```

Effects: Initialises `value_` from `static_cast<decltype(x)>(x)`; *Returns:* An implementation-defined awaitable type.

```
template <class TVal, class TAlloc>
auto yield_value(elements_of<generator<Ref, TVal, TAlloc>> g) noexcept;
```

Mandates:

- `TAlloc` meets the `Cpp17Allocator` requirements,

Effects: Execution is transferred to the coroutine represented by `g.coroutine_` until its completion. After `g.coroutine_` completes, the current coroutine is resumed. If `g.coroutine_` completes with an exception, the exception is rethrown from the 'co_yield' expression.

Variables with automatic storage duration in the scope of the coroutine represented by `g.coroutine_` are destroyed before variables with automatic storage duration in the scope of the coroutine denoted by this coroutine.

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

Returns: An implementation defined `awaitable` type which takes ownership of the generator `g`.

```
template<std::ranges::input_range R>
  requires convertible_to<ranges::range_reference_t<R>, Ref>
auto yield_value(elements_of<R> rng);
```

Effects: Equivalent to:

```
{
    auto it = std::ranges::begin(rng.get());
    auto itEnd = std::ranges::end(rng.get());
    while (it != itEnd) {
        co_yield *it;
        ++it;
    }
}
```

The return object is of an implementation defined type T such that `is_same_v<coroutine_traits<T>::promise_type, coroutine_traits<generator>::promise_type>` is `true`.

[*Note:* The coroutine state is allocated with `Allocator` - or `allocator<byte>` if `is_same_v<Allocator, void>` is `true`. — *end note*]

Returns: An implementation-defined awaitable type.

```
void unhandled_exception();
```

Effects: Equivalent to: `throw;`

```
static void* operator new(size_t size) requires same_as<Allocator, void>;
```

Effects: Allocates the coroutine state with `std::allocator`.

```
static void* operator new(size_t size) requires (!same_as<Allocator, void> && is_default_constructible_v<Allo
```

Effects: Allocates the coroutine state with a default-constructed instance of `Allocator`.

```
template<typeame Alloc, class... Args>
static void* operator new(size_t size, allocator_arg_t, Alloc& alloc, Args&...);

template<class This, typeame Alloc, class... Args>
static void* operator new(size_t size, This&, allocator_arg_t, Alloc& alloc, Args&...);
```

Mandates:

- `same_as<Allocator, void> || convertible_to<Alloc, Allocator>` is `true`,
- `Alloc` meets the `Cpp17Allocator` requirements,

Effects: Allocates the coroutine state with `Alloc`.

[*Note:* If `std::allocator_traits<Alloc>::is_always_equal::value` is `false` or if `is_default_-constructible_v<Alloc>` is `false`, `alloc` is stored in the allocation for the coroutine state. — *end note*]

```
static void operator delete(void* pointer, size_t size) noexcept;
```

Deallocate the coroutine state with an instance of the allocator equivalent to the one that was use to allocate it.

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

```
template <class Ref, class Value, class Allocator>
class generator<Ref, Value, Allocator>::iterator {
private:
    std::coroutine_handle<promise_type> coroutine_; // exposition only

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;

    iterator& operator=(iterator&& other) noexcept;

    explicit iterator(std::coroutine_handle<promise_type> coroutine) noexcept; // exposition only

    bool operator==(default_sentinel_t) const noexcept;

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

    reference operator*() const noexcept(std::is_nothrow_copy_constructible_v<reference>);

};
```

`iterator(iterator&& other) noexcept;`

Effects: Initializes `coroutine_` with `exchange(other.coroutine_, {})`.

`iterator& operator=(iterator&& other) noexcept;`

Effects: Equivalent to `coroutine_ = exchange(other.coroutine_, {});`

`explicit iterator(std::coroutine_handle<promise_type> coroutine) noexcept;`

Effects: Initializes `coroutine_` with `coroutine`.

`bool operator==(default_sentinel_t) const noexcept`

Returns: `!coroutine_ || coroutine_.done()`.

`iterator& operator++();`

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

Effects: Equivalent to:

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

`void operator++(int);`

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

Effects: Equivalent to:

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

```
reference operator*() const
noexcept (noexcept(std::is_nothrow_copy_constructible_v<reference>));
```

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

Effects: Equivalent to:

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

Feature test macros

Insert into [version.syn]

```
#define __cpp_lib_generator <DATE OF ADOPTION> // also in <ggenerator>
```

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] Corentin Jabot, Eric Niebler, and Casey Carter. P1206R3: ranges::to: A function to convert any range to a container. https://wg21.link/p1206r3, 11 2020.

[4] Gor Nishanov. P1681R0: Revisiting allocator model for coroutine lazy/task/generator. https://wg21.link/p1681r0, 6 2019.

[5] 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/T58h1W

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