std::generator: Synchronous Coroutine Generator for Ranges

Document #: P2502R0
Date: 2021-12-13
Project: Programming Language C++
Audience: LEWG
Reply-to: Casey Carter <Casey@Carter.net>

Abstract

We propose a standard library type std::generator which implements a coroutine generator that models std::ranges::input_range.

Acknowledgements

I'd like to thank Lewis Baker and Corentin Jabot, whose P2168 [3] I looted shamelessly for this proposal. This paper is presented as a new revision of their unpublished D2168R4 for continuity of design.

Revisions

P2502R0

• Reorder generator’s template parameters. This allows the reference type to be more easily defaulted to a true reference, while still respecting requests for differing value and reference types. This preserves the previous design’s ease-of-use, while providing full generality.

• Remove concerns about the $O(1)$ destruction requirement for view, which has been relaxed by P2415R2 “What is a view?” [7].

D2168R4

• Wording improvements

P2168R3

• Wording improvements
P2168R2

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

P2168R1

• 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

```cpp
std::generator<int> fib() {
    auto a = 0, b = 1;
    while (true) {
        co_yield std::exchange(a, std::exchange(b, a + b));
    }
}

int answer_to_the_universe() {
    auto rng = fib() | std::views::drop(6) | std::views::take(3);
    return std::ranges::fold_left(std::move(range), 0, std::plus{});
}
```

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 recursive generator is non-trivial; the standard should provide one.

Design

While the proposed `std::generator` interface is fairly straightforward, a few decisions are worth pointing out.
input_view

std::generator is a move-only view which models input_range and has move-only iterators. This is because the coroutine state is a unique resource (even if the coroutine handle is copyable).

Header

Multiple options are available as to where to 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>`

This paper uses a new `<generator>` header since P2168R3 did so, and LEWG has provided no guidance to do otherwise. We do note on our MSVC STL branch implementation that `#include <ranges>` includes 52.6k lines of code, and `#include <generator>` 53.3k lines. (Note that `<generator>` is specified to include both `<ranges>` and `<coroutines>`.) Defining generator in `<ranges>` together with a `#include <coroutine>` would penalize people who want `<ranges>` but not generator by about 610 LoC.

Reference type

generator has 3 template parameters: generator<T, Allocator = void, U = void>

From T and U, we derive types:

Using Value = conditional_t<is_void_v<U>, remove_cvref_t<T>, U>;

Using Reference =
conditional_t<is_void_v<U>,
conditional_t<is_reference_v<T>, T, const T&>,
T>;

Using Yielded = conditional_t<is_reference_v<Reference>, Reference, const Reference&>;

- Value is a cv-unqualified object type that specifies the value type of the generator's range and iterators,
- Reference specifies the reference type (not necessarily a core language reference) of the generator's range and iterators, and
- Allocator is the type of allocator used for the coroutine state, which can be void to type-erase any allocator specified as a coroutine argument, defaulting to allocator<byte> when none is specified.
- Yielded (necessarily a reference type) is the type of the parameter to the primary overload of yield_value in the generator's associated promise type.
Our expectation is that 98% of use cases will need to specify only one parameter. The resulting generator:

- has a value type of `remove_cvref_t<meow>`
- has a reference type of `meow`, if it is a reference type, or `const meow&` otherwise,
- expects `co_yield` to appear in the body of the generator with operands that are convertible to the reference type, and
- can use any allocator (via type-erasure) defaulting to `allocator<byte>`.

This avoids the performance pitfall from earlier revisions of the proposal that used the first argument type directly as reference type; users who naively chose `generator<std::string>` got an iterator that produces independent copies of the yielded value on every dereference, when they may have been satisfied by yielding a reference to the same constant value.

For the additional 1.5% of use cases that want to specify an allocator type statically so they need not constantly pass pairs of `allocator_arg`, `my_allocator` arguments to every coroutine, we have two-argument `generator`. The resulting generator:

- has a value type of `remove_cvref_t<meow>`
- has a reference type of `meow`, if it is a reference type, or `const meow&` otherwise,
- expects `co_yield` to appear in the body of the generator with operands that are convertible to the reference type, and
- can use any allocator convertible to `woof`, defaulting to a default-constructed `woof` if it is `default_initializable`.

Other than the change in allocation behavior, two-argument `generator` is identical to, and therefore as easy-to-use as, one-argument `generator`.

For the last 0.5% of people who need `generator` to step outside the box and use a proxy reference type, or who need to generate a range whose iterators yield prvalues for whatever reason, we have three-argument `generator`. The resulting generator:

- has a value type of `quack`,
- has a reference type of `meow`,
- expects `co_yield` to appear in the body of the generator with operands that are convertible to `meow`, if it is reference type, and otherwise `const meow&`, and
- can use any allocator (via type-erasure) if `woof` is `void`, or otherwise can use any allocator convertible to `woof`. 
Your iterators can yield a prvalue, but it must be a `copy_constructible` type so a copy of the operand of a single `co_yield` can be returned multiple times from repeated dereferences of the same iterator value.

**Obsolete discussion about reference specification**

[Note: Before P2502R0, generator’s first parameter `Type` denoted the reference type of the range / iterators, and the value type was defaulted to `remove_cvref_t<Type>`. The following sections of design discussion from that era are preserved here. — end note]

In earlier versions of this paper, the reference type was exactly the first template parameter. This had the advantage of being simple. But it was a terrible performance trap:

Consider the behavior of the following code assuming the reference type is exactly the first template argument:

```cpp
std::generator<std::string> f() {
    std::string hello = "hello";
    co_yield hello; // 0 or 1 copy depending on implementation
    co_yield "Hello"; // 1 copy (conversion from const char* to std::string)
}
for (auto&& str : f()) {} // 1 copy (*it returns std::string)
```

Of course the solution, which we advocated for, is for the user to manually specify an explicit reference type:

```cpp
generator<const std::string&> f() {
    std::string hello = "hello";
    co_yield hello; // 0 or 1 copy depending on implementation
    co_yield "Hello"; // 1 copy (conversion from const char* to std::string)
}
for (auto&& str : f()) {} // 0 copy
```

This works, can be explained, and is even logical. You get what you asked for. It is nonetheless surprising for non-experts that using the simple `generator<string>` would create 2 copies per `co_yield`.

To hope users would not routinely forget to use a reference type when using `std::generator` calls for a heaping barrel of optimism.

We later proposed that for a `generator<T>`, its reference type be `conditional_t<is_reference_v<T>, T, const T&>`.

<table>
<thead>
<tr>
<th>First parameter</th>
<th>reference type</th>
<th>default value</th>
<th>can yield mutable lvalue ref?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>const int&amp;</code></td>
<td><code>int</code></td>
<td>No</td>
</tr>
<tr>
<td><code>const int&amp;</code></td>
<td><code>const int&amp;</code></td>
<td><code>int</code></td>
<td>No</td>
</tr>
<tr>
<td><code>int&amp;</code></td>
<td><code>int&amp;</code></td>
<td><code>int</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>int&amp;&amp;</code></td>
<td><code>int&amp;&amp;</code></td>
<td><code>int</code></td>
<td>No</td>
</tr>
<tr>
<td><code>const int&amp;&amp;</code></td>
<td><code>const int&amp;&amp;</code></td>
<td><code>int</code></td>
<td>No</td>
</tr>
</tbody>
</table>
Attempts have been made to characterize the exact relations between reference, value, storage, and \texttt{co\_yield} exception types and categories. Ultimately, a simpler mental model is to characterize what expressions can be yielded for a given reference type and how many copies are made for each scenario.

<table>
<thead>
<tr>
<th>First parameter</th>
<th>\texttt{co_yield const T&amp;}</th>
<th>\texttt{co_yield T&amp;}</th>
<th>\texttt{co_yield T&amp;}</th>
<th>\texttt{co_yield U&amp;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{T}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{const T&amp;}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{T&amp;}</td>
<td>Ill-formed</td>
<td>0</td>
<td>Ill-formed</td>
<td>Ill-formed</td>
</tr>
<tr>
<td>\texttt{T&amp;&amp;}</td>
<td>Ill-formed</td>
<td>Ill-formed</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>\texttt{const T&amp;&amp;}</td>
<td>Ill-formed</td>
<td>Ill-formed</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

In this table, we see that only \texttt{co\_yield} that requires conversion incurs copy, which is expected. Coroutines guarantee that the yielded expression exceeds the lifetime of the \texttt{co\_yield} expression, so \texttt{generator} can usefully store a pointer to the object denoted by a yielded xvalue.

c\texttt{co\_yield} expressions involving conversion can store the yielded value in an awaiter. The type of the stored expression is the reference type with its reference qualifiers stripped, but that is an implementation detail that is not observable and is therefore of limited interest. Of course, that type needs to be constructible from yielded values.

Besides the \texttt{T} case, this behaves very much like returning from a function that is intended.

**Move-only and immovable types**

LEWG was interested in how this works with \texttt{generator} of move-only and immovable types.

<table>
<thead>
<tr>
<th>First parameter</th>
<th>\texttt{co_yield const T&amp;}</th>
<th>\texttt{co_yield T&amp;}</th>
<th>\texttt{co_yield T&amp;&amp;}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{move_only}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{const move_only&amp;}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{move_only&amp;}</td>
<td>Ill-formed</td>
<td>0</td>
<td>Ill-formed</td>
</tr>
<tr>
<td>\texttt{move_only&amp;&amp;}</td>
<td>Ill-formed</td>
<td>Ill-formed</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{const move_only&amp;&amp;}</td>
<td>Ill-formed</td>
<td>Ill-formed</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{immovable}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{const immovable&amp;}</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{immovable&amp;}</td>
<td>Ill-formed</td>
<td>0</td>
<td>Ill-formed</td>
</tr>
<tr>
<td>\texttt{immovable&amp;&amp;}</td>
<td>Ill-formed</td>
<td>Ill-formed</td>
<td>0</td>
</tr>
<tr>
<td>\texttt{const immovable&amp;&amp;}</td>
<td>Ill-formed</td>
<td>Ill-formed</td>
<td>0</td>
</tr>
</tbody>
</table>

As that table shows, these types work exactly like other types. However, to be able to move from a move only reference type, the coroutine has to explicitly state so:

```cpp
auto f = []() -> std::generator<move\_only> { co\_yield move\_only{}; }();
for (auto\&& x : f) {
    move\_only mo = std::move(x); // ill-formed, decltype(x) is const move\_only\&
}
```

```cpp
auto f = []() -> std::generator<move\_only\&&> { co\_yield move\_only{}; }();
for (auto\&& x : f) {
```

6
move_only mo = x; // ok
}

auto f = []() -> std::generator<move_only&> { move_only m; co_yield m; }();
for (auto& x : f) {
    move_only mo = std::move(x); // dicey but okay
}

Potential downsides

auto f = []() -> std::generator<MyType> {
    MyType t;
    co_yield std::move(t);
}();

In the example above std::move doesn't move. Arguably more than usual. Indeed the code expands to something similar to:
auto&& __temp = std::move(t);
yield_value(__temp); // <=> promise.value = std::addressof(__temp); // no move

Of course, a move would not have occurred for a std::generator<const MyType&> either as these things are identical. It might be surprising? The only way to avoid that is to create temporary value for rvalue reference, which would force a move to actually occurs, at the cost of performance.

Alternatives considered

Mandating a reference as the first parameter  We could make generator<int> ill-formed and force people to specify a reference type like generator<const int&>. We do not think this is very user-friendly, given that we can provide a reasonable default.
We rejected this option.

Using T& as the default  There are two issues with mutable references:

  • They are mutable (They allow mutating the coroutine frame), which would be an interesting default.
  • They are very restrictive as to the set of co_yield expression allowed with them.

We rejected this option.

Using T&& as the default  This avoids a copy when doing auto object = *it (where it is a std::generator::iterator), but it is very easy to misuse, consider:

auto f = []() -> std::generator<std::string> { co_yield "footgun"; }();
for (auto&& x : f) {
    auto y = x; // nothing suggest a move
    y.transform();
if (x != y) {
    // always triggers, likely to be surprising
}

We rejected this option.

Doing something clever for move-only types We considered returning \(T\) for move-only types so that they can be moved from by default. We realized this was too clever and inconsistent. Notably, adding a copy constructor to \(T\) would change the meaning of the code.

We rejected this option.

Doing something clever for reference types By default \(\text{generator}<\text{reference\_wrapper}<T>\) could yield \(\text{reference\_wrapper}<T>\) has that is already a "reference-like" type. However, no other view does that, "reference-like" is fuzzily defined, and this would probably cause more trouble than it's worth.

We rejected this option.

Keeping the D2168R4 design Returning values has the potential to severely impact performance, is inconsistent with other views, and is not necessary. It also did not work with move-only types.

The change, along with an implementation strategy described in the “How to store the yielded value in the promise type?” guarantees that no copy needs to be made if the reference and yielded types are the same (with qualifiers stripped).

We think this new approach keeps the simplicity of the original design, improves performance, and works with more types.

Thank you LEWG, and in particular Mathias, for highlighting these concerns!

Separately specifyable Value Type

This proposal supports specifying both the “yielded” type, which is the iterator’s 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 \(\text{zip}\):

```cpp
namespace ranges = std::ranges;

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

In this second example, using string as value type ensures that calling code can take the necessary 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, void, std::string_view> string_views() {
    co_yield "foo";
    co_yield "bar";
}

std::generator<std::string_view, void, 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 behavior operating on elements of v that were invalidated while iterating through the generator.
auto v = std::ranges::to<vector>(strings()); // (P1206R3 [4])

How to store the yielded value in the promise type?

There are multiple implementation strategies possible to store the value in the generator. An early revision of this paper always stored a copy of the yielded value, leading to an extra copy. Later revisions supported storing the yielded value in an awaitable object returned from the promise's yield_value function.

However, the object denoted by a glvalue yield expression is guaranteed to live until the coroutine resumes. We can take advantage of that fact by storing only a pointer to in the promise, if the result of dereferencing that pointer is convertible to the generator’s reference type. We guarantee this is the case by providing a yield_value whose parameter type is conditional_t<is_reference_v<Reference>, Reference, const Reference&>. This forces any conversions to happen inside the coroutine itself, yielding a temporary glvalue that can later be dereferenced to an lvalue which is trivially static_casted to Reference in the iterator's operator*.
A drawback of this solution is that the yielded value is only destroyed at the end of the full expression in which co_yield appears, so given

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

We think this is a reasonable tradeoff given that this approach minimizes the number of copies must be made of the yielded value. We force the coroutine to materialize the element to be yielded, but after doing so can cleanly pass a reference to that element through the coroutine and iterator machinery and directly to consuming code.

**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

```cpp
std::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);
    }
}

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

Example: A generator can also be used recursively

```cpp
using namespace std;

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

generator<int> visit(Tree& tree) {
    if (tree.left) co_yield ranges::elements_of(visit(*tree.left));
    co_yield tree.value;
    if (tree.right) co_yield ranges::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 $O(\text{depth})$ coroutine resumptions/suspensions per element of the sequence.

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

```cpp
using namespace std;

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

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

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

// nested_ints() is semantically equivalent to the following:
std::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
}

std::ranges::elements_of

ranges::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> yields the value directly, and

• co_yield elements_of(<expression>) yields successive elements 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

    std::ranges::elements_of(x)

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 two extra pointers per generator\(^1\).

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 [8] optimizations are possible.

\(^1\)The two pointers in our implementation have non-overlapping active times; we believe the pair can be optimized into a single pointer's space with some bit hacking to store a discriminator in the unused lower bits.
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 a 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_generators.

That third option would make the following ill-formed:

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

Instead you would need to write:

```cpp
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 the section on allocators).

**Allocator support**

In line with the design exploration done in section 2 of P1681R0 [6], std::generator supports both stateless and stateful allocators and strives 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 [6].
coroutine_parameter_preview_t such as discussed in section 3 of P1681R0 [6] has not been explored in this paper.

```cpp
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(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)**

```cpp
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, const int&, void> where void denotes that the allocator is type erased. The allocator is stored in the same allocation as 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**

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

f();
```

Again, returns a generator of type std::generator<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, but the default allocator (std::allocator) need not be stored since it is stateless.

**Explicit stateless allocator**
std::generator<int, std::stateless_allocator<int>> f() {}
f();

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

Explicit stateful allocator

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

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

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 [6] later.

Interaction of symmetric transfer and allocator support

The allocator must necessarily be part of a coroutine's promise type since implementations query the promise for allocation functions. Nonetheless, it would seem silly for a generator to be unable to nest another generator with identical element type but differing allocator. For that matter, even differing value types shouldn't be problematic: the only interface between the generator and the coroutine it wraps that differs depending on the type arguments to generator is yield_value. Ideally, generators would be able to recurse into other generators whose yield_value has the same parameter type even if all three template arguments to generator differ.

Our implementation uses a base class to implement the non-allocation behaviors for generator's promise so that generators with different allocator types can yield each other. Doing so, however, requires that we partially erase the type of a coroutine_handle so we can resume it later knowing only that its promise type derives from a particular base.

There are at least two ways to implement this partial type erasure:

- Storing a pointer in the common base to a component with full type knowledge, which can then resume the targeted coroutine,
- Relax the preconditions on some of the coroutine_handle functions to allow conversion from coroutine_handle<void> to coroutine_handle<T> when the source's corresponding address() value was obtained from a coroutine_handle referring to a coroutine whose promise object is pointer-interconvertible with an object of type T.
Our current plan is to standardize the intent to allow yielding nested generators with different allocator and value types, leaving the details of the implementation unspecified, and to later separately propose the changes to coroutine_handle that enable that implementation to be maximally efficient.

**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 - it notably works well with three different major standard libraries.

**Performance & benchmarks**

[Note: These benchmark results are fairly dated now - roughly a year old - and should be taken with a grain of salt. — end note]

Because implementations are still being perfected, and because performance is extremely dependant on whether HALO optimization (see P0981R0 [8]) 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.

<table>
<thead>
<tr>
<th></th>
<th>Clang</th>
<th>Clang ST</th>
<th>GCC</th>
<th>GCC ST</th>
<th>MSVC</th>
<th>MSVC ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single value</td>
<td>(1) 0.235</td>
<td>(2) 2.36</td>
<td>12.4</td>
<td>13.4</td>
<td>61.9</td>
<td>63.7</td>
</tr>
<tr>
<td>Single value, noinline (3)</td>
<td>13.5</td>
<td>13.7</td>
<td>14.1</td>
<td>15.2</td>
<td>63.8</td>
<td>64.4</td>
</tr>
<tr>
<td>Deep nesting</td>
<td>43670266.0</td>
<td>(4) 427955.0</td>
<td>58801348</td>
<td>338736</td>
<td>224052033</td>
<td>4760914</td>
</tr>
</tbody>
</table>

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 approaches is similar (3).
- Supporting recursion is greatly beneficial to nested/recursive algorithms (4).

The code for these benchmarks, as well as more detailed results, can be found on Github.
Wording

Drafting Note: Wording is relative to Working Draft N4901 [5].
Modify [ranges.general] as follows:

**General**

This Clause describes components for dealing with ranges of elements.
The following subclauses describe range and view requirements, and components for range primitives and range generators as summarized in Table [tab:range.summary].

Drafting Note: Add a new row at the end of [tab:range.summary] “Range generators” with header <generator> referring to the new subclause [coroutine.generator] added below.

Drafting Note: Add the declaration of ranges::elements_of to the <ranges> synopsis:

### Header <ranges> synopsis

```cpp
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;

    // [range.dangling], dangling iterator handling
    struct dangling;

    // [elementsof.overview], class template elements_of
    template<range R, class Allocator = allocator<byte>>
    class elements_of;

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

    [...]
}
```

Drafting Note: Insert the following new subclause immediately after [range.dangling]:

#### class template elements_of

Overview

Specializations of elements_of encapsulate a range and act as a tag in overload sets to disambiguate when a range should be treated as a sequence rather than a single value.

**Example:**

```cpp
    std::generator<any> f(std::ranges::input_range auto&& rng) {
        co_yield rng; // yield rng as a single value
```
namespace std::ranges {

    template<range R, class Allocator = allocator<byte>>
    class elements_of {
        private:
            [[no_unique_address]] Allocator allocator_; // exposition only
            R&& range_; // exposition only

        public:
            constexpr explicit elements_of(R&& r)
                noexcept(is_nothrow_default_constructible_v<Allocator>)
                requires default_initializable<Allocator>;

            constexpr explicit elements_of(R&& r, Allocator allocator) noexcept;

            constexpr R&& range() noexcept;

            constexpr Allocator get_allocator() const noexcept;
    };

    template<class R, class Allocator = allocator<byte>>
    elements_of(R&&, Allocator = {}) -> elements_of<R, Allocator>;

} // namespace std::ranges

---

Members

```cpp

constexpr explicit elements_of(R&& r)
    noexcept(is_nothrow_default_constructible_v<Allocator>)
    requires default_initializable<Allocator>;

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

constexpr explicit elements_of(R&& r, Allocator allocator) noexcept;

    Effects: Initializes allocator_ with std::move(allocator) and range_ with std::forward<R>(r).

constexpr R&& range() noexcept;

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

constexpr Allocator get_allocator() const noexcept;

    Returns: allocator_.
```

Drafting Note: Add the following subclause to the end of [ranges]:

**Range Generators**

**Overview**

generator presents a view of the elements yielded by the evaluation of a coroutine.

A 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 statement or the end of the coroutine. Elements of the sequence are produced by the coroutine each time a co_yield statement is evaluated. When the co_yield statement is of the form co_yield elements_of(rng), each element of the range rng is successively produced as an element of the generator.

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

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

— end example

**Header <generator> synopsis**

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

namespace std {
    // [coroutine.generator.class], class template generator
template<class T, class Allocator = void, class U = void>
    class generator;

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

**Class template generator**

```cpp
namespace std {
    template<class T, class Allocator = void, class U = void>
    class generator {
        using value = // exposition only
            conditional_t<is_void_v<U>, remove_cvref_t<T>, U>;
        using reference = // exposition only
            conditional_t<is_void_v<U>,
```
```cpp
conditional_t<is_reference_v<T>, T, const T&>,
T>;
using yielded = // exposition only
conditional_t<is_reference_v<reference>, reference, const reference&>;
class iterator; // exposition only

class promise_type;

generator(const generator&) = delete;
generator(generator&& other) noexcept;
~generator();
generator& operator=(const generator&) = delete;
generator& operator=(generator&& other) noexcept;

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

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

coroutine_handle<promise_type> coroutine_ = nullptr; // exposition only
};

Mandates: value is a cv-unqualified object type.

Mandates: reference is either a reference type, or a cv-unqualified object type that models copy_constructible.

Allocator shall be void, or shall either meet the Cpp17Allocator requirements.

Mandates: Let RRef denote remove_reference_t<reference>&& if reference is a reference type, or reference otherwise. Each of:

• common_reference_with<reference&&, value&>,
• common_reference_with<reference&&, RRef&&>, and
• common_reference_with<RRef&&, const value&>

is modeled. [Note: These requirements ensure the exposition-only iterator type can model indirectly_readable and thus input_iterator. — end note]

Specializations of generator model view and input_range.

The behavior of a program that defines a partial or explicit specialization of generator is undefined.

An instance of generator has an associated stack of coroutines, which is initially empty. A coroutine is associated with at most one generator instance at a given time.

21
Members

explicit generator(coroutine_handle<promise_type> coro) noexcept;

Initializes coroutine_ with coro.

generator(generator&& other) noexcept;

Initializes coroutine_ with exchange(other.coroutine_, {}).~generator();

Effects: Equivalent to:
   if (coroutine_) {
      coroutine_.destroy();
   }
generator& operator=(generator&& that) noexcept;

Effects: Equivalent to:
   if (auto old = exchange(coro, exchange(that.coro, {}))) {
      old.destroy();
   }

Returns: *this.

iterator begin();

Preconditions: coroutine_ refers to a coroutine suspended at its initial suspend-point.

Effects: Equivalent to:
   coroutine_.resume();
   return iterator(coro);

Remarks: This function pushes coroutine_ onto the generator’s empty stack of associated coroutines.

[Note: A program that calls begin more than once on the same generator has undefined behavior. — end note]
default_sentinel_t end() const noexcept;

Returns: default_sentinel.

class generator::promise_type

template<class T, class Allocator, class U>
class generator<T, Allocator, U>::promise_type {
   friend generator;

   add_pointer_t<yielded> value_ = nullptr; // exposition only
public:
    generator get_return_object() noexcept;
    suspend_always initial_suspend() noexcept;
    auto final_suspend() noexcept;
    suspend_always yield_value(yielded value) noexcept;

    template<class T2, class Alloc2, class U2>
    requires same_as<typename generator<T2, Alloc2, U2>::yielded, yielded>
    auto yield_value(ranges::elements_of<generator<T2, Alloc2, U2>> g) noexcept;

    template<ranges::input_range R, class Alloc2>
    requires convertible_to<ranges::range_reference_t<R>, yielded>
    auto yield_value(ranges::elements_of<R, Alloc2> r) noexcept;

    void await_transform() = delete;
    void return_void() noexcept {}
    void unhandled_exception();

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

    template<class Alloc, class... Args>
    requires same_as<Allocator, void> || convertible_to<Alloc, Allocator>
    static void* operator new(size_t size, allocator_arg_t, Alloc&& alloc, Args&...);

    template<class This, class Alloc, class... Args>
    requires same_as<Allocator, void> || convertible_to<Alloc, Allocator>
    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 get_return_object() noexcept;
    
    Returns: generator{coroutine_handle<promise_type>::from_promise(*this)}.

suspend_always initial_suspend() noexcept;
    
    Returns: {}.

auto final_suspend() noexcept;
    
    Preconditions: The coroutine whose promise object is *this is at the top of the stack of associated coroutines of some generator instance x.
    
    Returns: An awaitable object of unspecified type whose member await_suspend removes
the coroutine whose promise is \*this from the top of x's stack of associated coroutines, and resumes execution of the new top-of-stack coroutine, if any.

```cpp
suspend_always yield_value(yielded x) noexcept;
```

**Effects:** Equivalent to: `value_ = addressof(x)`.

**Returns:** `{}`.

template<class T2, class Alloc2, class U2>
requires same_as<typename generator<T2, Alloc2, U2>::yielded, yielded>
auto yield_value(ranges::elements_of<generator<T2, Alloc2, U2>> g) noexcept;

**Preconditions:** The coroutine whose promise object is \*this is at the top of the stack of associated coroutines of some generator instance x.

**Returns:** An object of an unspecified awaitable type ([expr.await]) which takes ownership of the generator g, whose member `await_suspend` pushes `g.coroutine_.atop` the stack of coroutines associated with x before resuming execution of `g.coroutine_.atop` and whose member `await_resume` rethrows any exception captured by a call to `g.coroutine_.atop.promise()`'s member `unhandled_exception`.

**Note:** 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 whose promise object is \*this. — end note]

template<ranges::input_range R, class Alloc2>
requires convertible_to<ranges::range_reference_t<R>, yielded>
auto yield_value(ranges::elements_of<R, Alloc2> r) noexcept;

**Effects:** Equivalent to:

```cpp
auto nested = [](allocator_arg_t, Alloc2, auto* range_ptr)
    -> generator<yielded, Alloc2, ranges::range_value_t<R>> {
    for (auto& e : *range_ptr)
        co_yield static_cast<yielded>(std::forward<decltype(e)>(e));
};
auto& rng = r.range();
return yield_value(ranges::elements_of(nested(
    allocator_arg, r.get_allocator(), addressof(rng))));
```

```cpp
void unhandled_exception();
```

**Preconditions:** The coroutine whose promise object is \*this is at the top of the stack of associated coroutines of some generator instance x.

**Effects:** If the coroutine whose promise object is \*this is the sole element of x's stack of associated coroutines, equivalent to: `throw`. Otherwise, stores the result of `current_exception()` where it can later be retrieved and rethrown by the `await_resume` member of the awaitable object returned from the `yield_value` call that pushed this coroutine onto x's stack of associated coroutines.
static void* operator new(size_t size)
requires same_as<Allocator, void> || default_initializable<Allocator>;

Let A be allocator<void> if Allocator denotes void, or Allocator otherwise. Let BAlloc be
allocator_traits<A>::template rebind_alloc<U> where U denotes an unspecified type
whose size and alignment are both _STDCPP_DEFAULT_NEW_ALIGNMENT__.

**Effects:** Initializes an allocator of type BAlloc with A{}, and uses that object to allocate
the smallest number of blocks that provide sufficient storage for:

- a coroutine state of size size,
- if allocator_traits<BAlloc>::is_always_equal::value is false, space to store a copy
  of the allocator, and
- if Allocator denotes void, any additional state necessary to ensure that operator
  delete can later deallocate this memory block with an allocator equal to the allocator
  used here.

**Returns:** A pointer to the space allocated for the coroutine state.

template<class Alloc, class... Args>
requires same_as<Allocator, void> || convertible_to<Alloc, Allocator>
static void* operator new(size_t size, allocator_arg_t, Alloc&& alloc, Args&...);

Let A be allocator<void> if Allocator denotes void, or Allocator otherwise. Let BAlloc be
allocator_traits<A>::template rebind_alloc<U> where U denotes an unspecified type
whose size and alignment are both _STDCPP_DEFAULT_NEW_ALIGNMENT__.

**Effects:** Initializes an allocator of type BAlloc with A(std::forward<Alloc>(alloc)), and
uses that object to allocate the smallest number of blocks that provide sufficient storage for:

- a coroutine state of size size,
- if allocator_traits<BAlloc>::is_always_equal::value is false or default_initial-
  izable<BAlloc> is false, space to store a copy of the allocator, and
- if Allocator denotes void, any additional state necessary to ensure that operator
  delete can later deallocate this memory block with an allocator equivalent to the
  allocator used here.

**Returns:** A pointer to the space allocated for the coroutine state.

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

**Preconditions:** pointer was returned from an invocation of one of the above overloads of
operator new with a size argument equal to size.

25
**Effects:** Deallocates the block of allocator memory that includes the coroutine state denoted by pointer using an allocator equivalent to the one that was used to allocate it.

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

```cpp
template<class T, class Allocator, class U>
class generator<T, Allocator, U>::iterator {
public:
    using value_type = value;
    using difference_type = ptdiff_t;

    iterator(iterator&& other) noexcept;
    iterator& operator=(iterator&& other) noexcept;
    reference operator*() const noexcept(is_nothrow_copy_constructible_v<reference>);
    iterator& operator++();
    void operator++(int);
    bool operator==(default_sentinel_t) const noexcept;
private:
    friend class generator;
    explicit iterator(coroutine_handle<promise_type> coroutine) noexcept; // exposition only
    coroutine_handle<promise_type> coroutine_; // exposition only
};
```

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

```cpp
iterator(iterator&& other) noexcept;
```

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

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

**Preconditions:** `coroutine_.done()` is false, and `coroutine_`

Let `p` be the promise object of the coroutine at the top of the stack of coroutines associated with the generator whose stack of associated coroutines includes `coroutine_`.

**Effects:** Equivalent to:

```cpp
return static_cast<reference>(*p.value_);
```

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

**Preconditions:** `coroutine_.done()` is false.
**Effects:** Resumes the coroutine at the top of the stack of coroutines associated with the generator whose stack of associated coroutines includes `coroutine_`.

**Returns:** `return *this;`

`void operator++(int);`

**Preconditions:** `coroutine_.done()` is false.

**Effects:** Equivalent to: `+++this`.

`bool operator==(default_sentinel_t) const noexcept;`

**Returns:** `coroutine_.done()`.

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

**Effects:** Initializes `coroutine_` with `coroutine`.

---

### Feature test macro

**Drafting Note:** Insert in lexicographical order in [version.syn] (updating YYYYXXL to the date of merge):

```
#define __cpp_lib_generator YYYYXXL // also in <generator>
```
References


[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] Casey Carter, Lewis Baker, Corentin Jabot std::generator implementation https://godbolt.org/z/oo75fTMc6