

**Document number:** P1063R0

**Date:** 2018-05-06

**Reply To:**

Geoff Romer ([gromer@google.com](mailto:gromer@google.com))

James Dennett ([jdennett@google.com](mailto:jdennett@google.com))

Chandler Carruth ([chandlerc@google.com](mailto:chandlerc@google.com))

**Audience:** WG21 Evolution Working Group

# Core Coroutines

Making coroutines simpler, faster, and more general

[Introduction](#)

[Non-goals](#)

[Proposed Design](#)

[Unwrap operator syntax](#)

[Unwrap operator semantics](#)

[Return type\(s\)](#)

[The suspend point parameter](#)

[Coroutine lambdas](#)

[Expository implementation](#)

[Coroutine functions](#)

[Tail calls](#)

[constexpr](#)

[Alternative: Patching the TS](#)

[Allocation and performance](#)

[API Complexity](#)

[Comparison](#)

[Conclusion](#)

[Acknowledgements](#)

[Appendix: Examples](#)

[Futures](#)

[Simple generator](#)

[Zero-allocation generator](#)

# Introduction

*“C++ is a general-purpose programming language providing a direct and efficient model of hardware combined with facilities for defining lightweight and efficient abstractions.*

Or terser:

*C++ is a language for developing and using elegant and efficient abstractions.”*

— Bjarne Stroustrup, *The C++ Programming Language (4th Edition)*

The Coroutines TS provides users with an elegant and efficient abstraction for writing asynchronous code. We mean that as both sincere praise, and as a critique: the Coroutines TS provides an abstraction, but it does not provide programmers with the facilities they need to define their own elegant and efficient abstractions. Furthermore, the TS’s abstraction prioritizes the asynchronous use case in a variety of ways, which prevents it from being general-purpose. It gives programmers ways of extending and reusing the asynchrony abstraction, but they remain locked into many of the design tradeoffs motivated by the original use case.

Fundamentally, the Coroutines TS does not provide a direct and efficient model of hardware<sup>1</sup>: the primitive objects and operations that are used to implement coroutines are hidden behind an abstraction boundary.

Nearly all of the most serious issues we identified in [P0973R0](#) are reflections of this problem:

- Programmers cannot reliably prevent coroutine-based code from allocating memory, even if they know the allocation is unnecessary, because the allocation takes place behind the abstraction boundary.
- Programmers cannot control variable-capture semantics, and can all too easily overlook them entirely, because the capture is hidden behind an interface that presents itself as a function call.
- The library bindings are extremely complex because different abstractions require the underlying primitives to be composed in different ways. The TS supports this by providing APIs for the programmer to *configure how they are composed*, rather than permitting the programmer to *write code that composes them*.
- The `co_await` keyword is an overt manifestation of the TS’s preference for the asynchronous use case.

---

<sup>1</sup> We suggest as a friendly amendment that the quote should say “a direct and efficient model of *the platform*”. For example, C++ templates provide a direct and efficient map of the compiler’s code generation facilities, rather than of any hardware feature.

In this paper, we propose exposing a minimal set of coroutine primitives that map directly to the underlying implementation. This results in a design for coroutines that is substantially simpler and yet can efficiently support a broader range of uses.

We see no practical way of making our proposed revisions backwards-compatible with the Coroutines TS design, so they must be adopted before coroutines reach an IS (if at all). Straw polling in Jacksonville indicated a strong desire to ship coroutines in C++20, but we are uncertain as to whether our proposal can be implemented and sufficiently vetted in the time remaining, so as an alternative we also present a much more minimal set of changes to the TS, which we believe are feasible in the C++20 timeframe, to address some of the concerns from P0973. However, these alternative changes necessarily avoid addressing the fundamental issue of hidden primitives, and instead focus on adding yet more configuration options to patch use cases already known to be problematic.

## Non-goals

This proposal is solely concerned with “stackless” coroutines, and does not address the kinds of problems that are solved by “stackful” coroutines. We fully support the committee’s decision to pursue stackless and stackful coroutines independently.

This proposal does not attempt to extend coroutines to be a fully general monad facility. For programmers who wish to adopt a monadic approach, both our proposal and the TS are limited to supporting *linear monads*, because they do not support copying a suspended coroutine frame, and consequently do not support nondeterministically resuming from the same state with multiple inputs.

## Proposed Design

### Unwrap operator syntax

We propose replacing the `co_await` keyword with an operator-like token, which we tentatively suggest spelling `[<-]` (we are very open to committee feedback on the spelling):

```
optional<string> f();  
string s = [<-] f();
```

```
future<string> g();  
string s = [<-] g();
```

```
expected<string> h();  
string s = [<-] h();
```

Our proposed spelling is intended to suggest *unwrapping*, which we regard as the most central meaning of these expressions. Correspondingly, we propose to refer to them as *unwrap expressions* rather than “await expressions”, and we refer to the operand of an unwrap expression as a *wrapper* (and its type as a *wrapper type*).

An operator-like token has two major advantages over an English-derived keyword:

- An operator can more easily avoid tying itself to a particular use case, as `co_await` is tied to asynchrony.
- An operator need not choose between colliding with existing identifiers in user code, or being so awkwardly spelled that no existing code uses it. It must avoid colliding with existing C++ syntax, but that’s a far more manageable problem.

**Option:** We could also introduce a binary operator analogous to `->`, such that `x op y` is equivalent to `([<-]x).y`. This would make it easier to chain applications of the unwrap operator.

**Alternative:** The unwrap token could be a suffix, rather than a prefix. This has the advantage of naturally supporting chaining:

```
optional_struct[->].optional_sub_struct[->].field
```

However, this would depart from C++ convention (unary operators are generally prefixes), and could reduce readability by making the token less prominent.

**Alternative:** A different keyword spelling could be less use-case-specific than `co_await`. However, any keyword will still suffer from the need to avoid collisions with identifiers, and it is doubtful if any keyword can fully capture the breadth of possible use cases. Our best suggestion along these lines would be something like `co_unwrap`, but unwrapping is not the sole meaning of this operation; just the most central one. For example, it’s a poor fit for unidirectional generators:

```
co_unwrap std::yield(foo); // Huh?
```

## Unwrap operator semantics

Consider how an expression `co_await x` is evaluated: the state of the enclosing coroutine is reified as an object, and passed to an algorithm that is controlled by the library associated with `x`. That algorithm may eventually do two things:

1. return a value to the coroutine’s caller, and
2. specify the value of the `co_await` expression, and resume the coroutine by invoking it.

The first is mandatory and synchronous, whereas the second is optional and may be synchronous or asynchronous.

Notice that both suspension and resumption of the coroutine act as inversions of control; this is most obvious in the case of resumption, where control returns from an expression via a function call, rather than a `return` statement, but the initial transfer of control (from a stack frame to an algorithm that takes that frame as input) is effectively a complementary inversion of control.

We propose to allow the library to implement that entire algorithm directly in C++ code, as a function which takes a coroutine object and returns the value that is returned to the coroutine's caller, while arranging for the coroutine to be resumed in whatever manner is appropriate to the library. This change is motivated by the observation that C++ code is a far simpler and more general way to specify an algorithm than overloading the ~15 extension points of a fixed algorithm specified by the standard.

Specifically, we propose specifying the algorithm by overloading `operator[<-]`. Here's an example of such an overload for `expected<T,E>`:

```
template <typename T, typename U, typename E, typename Coroutine,
          typename SuspendPoint>
expected<U, E> operator[<-](
    const expected<T,E>& e, Coroutine& coroutine, SuspendPoint s,
    implicit_convert<expected<U, E>>& final_return) [<-] const T& {
    if (e.has_value()) {
        tail return coroutine(*e, s, final_return);
    } else {
        return unexpected(e.error());
    }
}
```

(See [below](#) for a discussion of `tail return`)

An `operator[<-]` overload's arguments are as follows (in order):

1. The object being unwrapped (this is the implicit object argument, if the operator is defined as a member).
2. The coroutine that contains the unwrap expression being executed.
3. A tag representing the point at which the coroutine was suspended (i.e. the specific unwrap expression being evaluated).
4. The "final return functor" that was passed into the coroutine when it was previously resumed. In addition to its role inside the coroutine (discussed below), this object can be used as a side channel to propagate information from one suspension to the next.

`operator[<-]`'s return value becomes the value returned to the caller who most recently resumed the coroutine. The coroutine can be resumed again by invoking it; that invocation takes arguments representing the value of the unwrap expression, the suspend point where it is currently suspended, and a final return functor. The final return functor is used to handle the coroutine's final return value; implicit and explicit `return` statements in the coroutine definition are transformed into invocations of the final return functor, and its nested member `return_type` determines the return type of the coroutine (we can't use `decltype` for this, because the functor

may be invoked with different argument types in different contexts). The final return functor also acts as a useful hook for ADL: because the functor is generally specified when the coroutine is created, it enables different kinds of coroutines to select different `operator[<-]` overloads, even if the wrapper type is the same.

Thus, the example above says that if `e` holds a `T` value, the coroutine is resumed, with `*e` as the value of the unwrap expression. This happens synchronously as part of the `[<-]` operation, so even though the coroutine returns when the `[<-]` operation returns, the effect is as if the `[<-]` operation simply returned control to the coroutine, which then returns normally. On the other hand, if `e` holds an error, that error is returned immediately to the coroutine's caller, and consequently the coroutine returns immediately, and the remainder of the coroutine is never executed.

In this case, the final return functor type is a simple generic helper that implicitly converts its argument:

```
template <typename T>
struct implicit_convert {
    using return_type = T;
    template <typename U> T operator()(U&& u) { return std::forward<U>(u); }
};
```

(Like all libraries discussed in this paper, this is merely an example; we are not proposing it for standardization at this time.)

Note that when evaluating an unwrap expression, control leaves the enclosing coroutine (without exiting any scopes) before the `operator[<-]` overload is invoked. Consequently, if an `operator[<-]` overload throws an exception, the coroutine will not be found during stack unwinding. An unwrap expression can only throw if its operand throws, or the coroutine is explicitly resumed with an exception via the `raise()` operation discussed [below](#).

We deliberately speak in terms of “unwrap expressions” rather than “the `[<-]` operator”, because the `[<-]` token doesn't precisely behave as an operator- it affects control flow, rather than merely being a syntactically-sugared function call. In this respect, `[<-]` draws on the precedent set by `new`.

## Return type(s)

Because of the aforementioned inversion of control, this function effectively has two return types: the type of the unwrap expression (in the domain of ordinary user code), and the type returned from the function body (in the inverted domain of the library). The purpose of the `[<-]` `const T&` suffix on the function signature is to specify the type of the unwrap expression, while ordinary return type syntax represents the inverted return type. Both types form part of its API (and hence part of its signature), and which one to treat as “the” return type will depend on the context. For example, the inverted return type is the return type for purposes of reading the

function body, which is our main motivation for having it occupy the return-type position in the signature. However, for purposes of name lookup and overload resolution, it makes more sense to treat the user return type as the return type, because the user return type is determined by the function implementation (although, unfortunately, not in a way that permits it to be easily deduced), whereas the inverted return type has to match the return type of the coroutine.

Indeed, for those purposes and for template argument deduction, we actually need the inverted return type to behave like a parameter, because `operator[<-]` effectively behaves like an implicit conversion from the operand type to the coroutine return type, so we need both types to participate in name lookup and resolution. This is one purpose of the `FinalReturn` parameter. It would be nice to avoid the duplication between the actual return type and `FinalReturn::return_type` (we are very open to suggestions on this point), but we note that this minimal burden falls only on authors of wrapper libraries, and we expect those to be relatively rare.

**Alternative:** The syntactic return type could represent the user return type, and the special syntax could represent the inverted return type. This would eliminate the need for the nested typedef `return_type` (and the associated duplication), at the cost of making the function body more surprising. Alternatively, we could require special syntax for both types, to minimize the risk of misinterpretation. We are very open to committee guidance on how to syntactically represent this rather odd situation.

**Alternative:** `operator[<-]` could be defined to return `void`, with the final return functor propagating its output via a library-defined side channel. This can be a useful technique in some cases (see e.g. the [future example](#)), but in others (such as the `expected` example we've just seen), it's much cleaner and simpler for `operator[<-]` to return a value.

Note that for simplicity, this example glosses over the issue of qualifiers on the `expected<T,E>` object: like `*e`, `[<-] e` should be mutable if and only if `e` is mutable, and should be an rvalue if and only if `e` is an rvalue. This can be accomplished via a set of four overloads (with the user return type qualified to match the parameter), and/or perfect forwarding (with the user return type computed via a metafunction such as [P0847R0's like\\_t](#)).

## The suspend point parameter

The suspend-point arguments will represent different suspend points with different types, so that the invocation of `coroutine` can be dispatched statically. However, the types are implicitly convertible to `size_t`, and `Coroutine` will be overloaded to take `size_t` and dynamically dispatch appropriately (e.g. via a `switch`). The suspend points, when converted to `size_t`, will be guaranteed not to equal `0` or `MAX_SIZE` so that both the implementation and the wrapper library have the option of using those values as sentinels for the beginning and end of execution.

Decoupling the suspend point from the coroutine has important performance benefits. In many cases (notably, generators) the library will need to persistently store a coroutine across many suspensions, tracking the suspend point dynamically. This design enables the library to track the suspend point dynamically, while retaining full static knowledge of the coroutine object itself. This in turn enables many optimizations. For example, with a dynamic suspend point and a static coroutine, the optimizer's existing constant folding and propagation logic should already be sufficient to eliminate the dynamic dispatch in many cases. On the other hand, if the coroutine is type-erased, eliminating the dynamic dispatch would essentially require devirtualization, a notoriously challenging form of optimization. These optimizations will be especially critical for generators, where the dispatch is more likely to be in a tight inner loop.

`coroutine` also provides a `raise()` method, which takes a `std::exception_ptr` and a dynamic suspend point<sup>2</sup>, and has the effect of resuming the coroutine as though the `[<-]` operation had thrown the given exception. Note that if the coroutine does not catch it, the exception will propagate back into the `raise()` caller.

The return type of `operator[<-]` must be convertible to the return type of `Coroutine`, because the return value of `operator[<-]` becomes the return value of the coroutine. Typically, the return type will be an instantiation of the same template as the wrapper, but that is not a requirement, and there are occasional use cases for deviating from that convention.

**Alternative:** We could instead have `operator[<-]` take a single continuation argument, which is callable with a single argument (the value callback) to resume the coroutine, and for which the suspension point is a hidden implementation detail. This would preclude the optimization discussed above (erasing the suspension point but not the coroutine), so we would probably need to permit wrapper types to provide overloads of either form, depending on whether they can take advantage of that optimization. This would provide one slight simplification: the suspend points for a given coroutine could be different values of a single type, since the two-argument form is only needed when the suspend point is going to be type-erased anyway, so the signature would be more like

```
template <typename T, typename E, typename U, typename E2, typename Coroutine>
expected<U, E2> operator[<-](
    const expected<T,E>& e, Coroutine& coroutine, size_t suspend_point,
    implicit_convert<expected<U, E2>>& final_return) [ <- ] const E&
```

However, permitting two syntactically and semantically different forms of `operator[<-]` overload seems like unnecessary complexity. Overloads of `operator[<-]` should be fairly rare, so there's relatively little benefit in making them easier to write, and there's a relatively high risk of inconsistency creating confusion.

---

<sup>2</sup> Static suspend point overloads could also be added, but we suspect that if you care about the cost of a single switch during exception handling, you're misusing exceptions.

## Coroutine lambdas

Since `operator[<-]` overloads take coroutine objects as inputs, we need a corresponding way to create coroutine objects. We propose to use lambda expression syntax for this purpose, since it has exactly the properties we need: it lets us define a callable object from a function body, and allows us to explicitly specify capture semantics. To distinguish coroutine lambdas from ordinary lambdas, we propose that coroutine lambdas should have the `do` keyword in place of the argument list:

```
future<string> foo();
future<int> bar();
...
auto my_coroutine = [] do -> future<int> {
    int i = ([<-]foo()).size();
    return i + [<-]bar();
};
```

A coroutine lambda is much like an ordinary lambda, except that its state includes not only its captures, but also any local variables that must be preserved across suspensions. Similarly, it exposes not only a call operator for the initial invocation, but also call operators and `raise()` methods for resuming execution, as described earlier.

Coroutine lambdas cannot take parameters. This is for reasons of safety: the code in a coroutine may continue executing after the initial function call has returned (from the caller's point of view), so if any temporary values were passed to pointer or reference parameters of the coroutine, they would be left dangling. The inputs to a coroutine lambda are instead expressed via the capture group. See [below](#) for how coroutine lambdas can be used to define ordinary functions with parameters, etc.

This design has two major benefits: first, it enables library code to control the creation, usage, and destruction of coroutine frames in exactly the same way as any other object (and in particular, allows the creation of coroutine libraries that are allocation-free by construction, rather than at the whim of the optimizer). Second, the capture syntax gives programmers explicit control over capture semantics (in the Coroutines TS, capture semantics are controlled by the parameter types, but parameter types are API-visible, and so API owners are not always at liberty to change them). Use of capture syntax also leverages programmers' existing intuitions: reference and pointer inputs to a coroutine are potentially hazardous in the same way, and for the same reasons, as the reference and pointer captures of an ordinary lambda.

Note that exceptions have no special semantics inside a coroutine: any exception that isn't caught in the body of the coroutine will propagate to the caller that resumed the coroutine (which will typically be library code associated with the coroutine, so this shouldn't have any major functional effects).

**Alternative:** we could specify that exceptions that escape the coroutine are forwarded via the final return functor. This would provide some minor benefits (primarily, greater consistency in how exceptions and ordinary returns are propagated), but also some minor drawbacks: it would complicate the API for the final return functor, and we would not be able to handle exceptions thrown from tail calls (i.e. unwrap expressions and return statements), which may be surprising, and may limit the consistency benefits.

## Expository implementation

The following example illustrates how a compiler might generate equivalent C++17 code for a given coroutine. Of course, this is not how we expect coroutine compilation to actually work, but it can serve as a “reference implementation” to understand the API and behavior of coroutine objects.

Consider the following code:

```
expected<string, Err> foo(const string& s);
expected<int, Err> bar();

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

The compiler could implement that by generating the following code:

```
// Convenience helper shared by all coroutine implementations
template <typename T>
class __manual_lifetime {
    std::aligned_storage_t<sizeof(T), alignof(T)> storage_;

public:
    template <typename... Args>
    void emplace(Args&&... args) {
        new (&storage_) (std::forward<Args>(args)...);
    }

    T& get() { return *reinterpret_cast<T*>(&storage_); }

    void destroy() {
        get().~T();
    }
};
```

```

class _f_1 {
public:
    _f_1(const _f_1&) = delete;
    _f_1(_f_1&&) = delete;
    _f_1& operator=(const _f_1&) = delete;
    _f_1& operator=(_f_1&&) = delete;

    // Beginning of execution
    template <typename FinalReturn>
    typename FinalReturn::return_type operator()(FinalReturn& final_return) {
        static_assert(
            std::is_same_v<typename FinalReturn::return_type, expected<int, Err>>);
        __tmp_1.emplace(foo(s));
        __suspend_point = 1;
        tail return operator[<-](__tmp_1.get(), *this, suspend_point_t<1>{}),
            final_return);
    }

    template <typename T, typename FinalReturn>
    typename FinalReturn::return_type operator()(
        [] -> T __val, size_t suspend_point, FinalReturn& final_return) {
        static_assert(
            std::is_same_v<typename FinalReturn::return_type, expected<int, Err>>);
        assert(suspend_point == __suspend_point);
        switch (suspend_point) {
            case 1:
                i.emplace(implicit_cast<unwrap_expression_type<1>>(
                    __val()).size());
                __tmp_1.destroy();
                __tmp_2.emplace(bar());
                __suspend_point = 2;
                tail return operator[<-](__tmp_2.get(), *this, suspend_point_t<2>{}),
                    final_return);

            case 2:
                int __result = i.get() + implicit_cast<unwrap_expression_type<2>>(
                    __val());
                __tmp_2.destroy();
                i.destroy();
                return final_return(__result);
        }
    }
}

// Statically dispatched version of the above. For ease of exposition, we're
// assuming the optimizer will inline, constant-fold, and eliminate the switch.
template <typename T, size_t suspend_point, typename FinalReturn>

```

```

typename FinalReturn::return_type operator()(
    [] -> T __val, suspend_point_t<suspend_point>, FinalReturn& final_return) {
    static_assert(
        std::is_same_v<typename FinalReturn::return_type, expected<int, Err>>);
    assert(n == __suspend_point);
    tail return (*this)(__val(), suspend_point, return_type);
}

template <typename FinalReturn>
typename FinalReturn::return_type raise(
    std::exception_ptr e, size_t suspend_point, FinalReturn&) {
    static_assert(
        std::is_same_v<typename FinalReturn::return_type, expected<int, Err>>);
    assert (suspend_point == __suspend_point);
    switch (suspend_point) {
        case 1:
            __tmp_1.destroy();
            std::rethrow_exception(e);
        case 2:
            __tmp_2.destroy();
            i.destroy();
            std::rethrow_exception(e);
    }
}

~_f_1() {
    switch(__suspend_point) {
        case 0: break;
        case 1:
            __tmp_1.destroy();
            break;
        case 2:
            __tmp_2.destroy();
            i.destroy();
            break;
    }
}

private:
// Implicitly invoked via lambda capture syntax
_f_1(const string& s) : s(s), __suspend_point(0) {}

template <size_t n>
using suspend_point_t = std::integral_constant<size_t, n>;

template <size_t n>
using unwrap_expression_type = /* see below */;

```

```

size_t __suspend_point;

// Captures
const string& s;

// Stack variables
//
// The layout of these members is purely illustrative; in practice we expect
// the compiler to lay out this class using the same algorithms it uses to
// lay out ordinary stack frames.
__manual_lifetime<int> i;

union {
    __manual_lifetime<expected<string, Err>> __tmp_1;
    __manual_lifetime<expected<int, Err>> __tmp_2;
};
};

```

Some notes on how we present this implementation:

- We rely on lazy function parameters, as proposed by [P0927R0](#), to avoid moving the unwrap expression result or storing it in the local stack frame (where it could inhibit tail call elimination). If lazy parameter support is unavailable, we can achieve the same effect by requiring `operator[<-]` overloads to wrap the expression result in a lambda before passing it in.
- The coroutine object tracks the suspend point solely as a way of tracking object liveness for use in the destructor (although as a convenience we also use it in debug diagnostics), because in that case we cannot require the suspend point to be supplied externally.
- For ease of exposition, we depict the dynamically-dispatched `operator()` as a single function template with a `switch` on the suspend point. This is not precisely accurate, because it would require all cases of the `switch` to compile for all argument types, but we only want to require that the `n`th case of the switch compiles when the argument type is `unwrap_expression_type<n>`; for any other argument type, its behavior is undefined.

The `operator()` overloads are never `const` (in effect, `do` implies `mutable`); in principle we could allow the user to specify or omit `mutable` as with ordinary lambdas, but in practice `mutable` would just be boilerplate, since it would only be correct and safe to omit it in cases where the coroutine has effectively no mutable stack variables, which we expect to be rare and marginal.

The `unwrap_expression_type<n>` alias represents the type of the `n`th unwrap expression, and hence the parameter type of the corresponding continuation.

If the return type was explicitly specified in the coroutine definition (which we expect to be rare), it must match the `return_type` of the final return functor (hence the `static_asserts` in the example code).

Note that the coroutine transformation does not affect the existing rules for the sequenced-before relation; if an unwrap expression and some other operation are unsequenced, the latter operation may be evaluated before the unwrap expression, or it may not be evaluated until the continuation is resumed (which, of course, may never happen).

## Coroutine functions

We expect that in most use cases, coroutine lambdas will not be part of public APIs; instead, they will be hidden implementation details of ordinary functions, which wrap the coroutine lambdas to handle issues such as parameter passing/capture, lifetime management, and whether to defer initial invocation of the lambda. Naively, these functions could require more boilerplate than coroutine functions in the TS, but we believe we can resolve that through a combination of relatively natural, orthogonal features, many of which are useful even outside of the context of coroutines.

As a motivating example, consider this asynchronous function:

```
// Consumes all bytes from `connection`, and returns the number
// of bytes consumed. `connection` must remain live until the returned
// future is ready.
auto count_bytes(Connection& connection) {
    return make_future<int>(new auto([&connection] do {
        int bytes_read = 0;
        vector<char> buffer(1024);
        while(!connection.done()) {
            bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
        }
        return bytes_read;
    }));
}
```

In this example `make_future` is a function provided by the `future<T>` library, which takes a pointer to a coroutine, takes ownership of it, invokes it, and returns a future representing the coroutine's result. The memory allocation is necessary because the coroutine object needs to outlive the initial `count_bytes` call.

The explicit memory allocation has at least three drawbacks:

- It's inconsistent with modern C++ style, which strongly discourages explicit `new`.

- It precludes the `make_shared`-style optimization of allocating the coroutine object together with the future's shared state.
- It adds boilerplate to the function definition.

We can solve the first two problems by having `make_future` take a callback, which it evaluates to obtain the coroutine object, allocating the result directly on the heap:

```
auto count_bytes(Connection& connection) {
    return make_future<int>([&] { return [&connection] do {
        int bytes_read = 0;
        vector<char> buffer(1024);
        while(!connection.done()) {
            bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
        }
        return bytes_read;
    }});
}
```

However, this doesn't address the boilerplate problem, and the addition of a second lambda makes the code even harder to read. However, this problem goes away if `make_future` takes a lazy parameter, as proposed in [P0927](#):

```
auto count_bytes(Connection& connection) {
    return make_future<int>([&connection] do {
        int bytes_read = 0;
        vector<char> buffer(1024);
        while(!connection.done()) {
            bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
        }
        return bytes_read;
    });
}
```

The boilerplate can be further reduced if we have a terse syntax for expression aliases (for example, by generalizing [P0573](#)'s proposed `=>` syntax to apply to functions):

```
auto count_bytes(Connection& connection) => make_future<int>(
    [&connection] do {
        int bytes_read = 0;
        vector<char> buffer(1024);
        while(!connection.done()) {
            bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
        }
        return bytes_read;
    });
```

We contend that this syntax contains almost no boilerplate other than a smattering of punctuation. The additional syntactic elements not present in the Coroutines TS all have important, user-facing functional roles:

- `make_future<int>` specifies what kind of coroutine this is, including (implicitly) the return type.
- The capture group specifies the capture semantics of the coroutine object.
- `do` acts as an introducer, specifying that the following block is a coroutine.

In all three cases, making these properties syntactically explicit has important advantages:

- The programmer has explicit, local control over what kind of coroutine is being defined, even if they do not control the function signature, e.g. because they must match an existing API (In the Coroutines TS, this can be controlled only via a trait parameterized by the parameter and return types). Symmetrically, the reader can easily tell what kind of coroutine they are reading.
- The programmer has explicit control over capture behavior, so that for example an argument can be captured by value (for safety) even if the API is obliged to pass by reference. Symmetrically, the capture behavior is explicitly visible in the code, cueing the reader (and programmer) to possible safety or performance concerns.
- The explicit introducer enables both the reader and the programmer to immediately and reliably recognize coroutine code. This eliminates the need for a separate `co_return` syntax to cue the compiler that it's processing a coroutine.

## Tail calls

Consider a coroutine like the following:

```
[&connection] do -> expected<int, Err> {
    int bytes_read = 0;
    vector<char> buffer(1024);
    while(!connection.done()) {
        bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
    }
    return bytes_read;
}
```

With the design described above, the unbounded iteration in this code will be transformed into an unbounded *recursion*, raising obvious concerns about stack size. However, the mutual recursion between `expected<T,E>::operator[<-]` and the generated coroutine code is actually all tail recursion, because every mutually recursive call is actually the final operation before the enclosing function returns. Consequently, the compiler should be able to apply tail call elimination (hereinafter “TCE”) to avoid growing the stack.

For this approach to be viable, programmers will need to have complete confidence that TCE will in fact be applied (even in non-optimizing build modes). There are two reasons this is difficult to achieve:

- There is currently no way to specify that TCE will take place, because the C++ standard has no explicit concept of stack storage as a finite resource.
- It's not as easy as it seems to determine whether a call is eligible for TCE in the first place. For example, a statement of the form `return f(...);` is nevertheless ineligible if there are any local variables with nontrivial destructors still live at that point (because then the function call is not actually the last operation before the return), or if the `f()` call takes a pointer or reference to any local variable. This is not an issue for `operator[<-]` calls inside the coroutine generated code (because the compiler can ensure that it's able to apply TCE to the code it generates), but it is an issue when user-defined `operator[<-]` overloads invoke the continuation synchronously.

To address the first issue, we propose adding standard wording such as the following:

“If this International Standard specifies that a function invocation is a *tail call*, then the implementation must disregard the invoking function call before entering the tail call, for purposes of enforcing any implementation-defined limits concerning the number of simultaneously active function calls, or the number or size of simultaneously-live variables with automatic storage duration. [Note: The effect of this requirement is that on implementations with a bounded stack, a tail call must reuse the stack frame of the calling function. — *end note*]

To address the second issue, we propose introducing a new syntax `tail return`, which requires its operand to be a tail call (`tail` is a contextual keyword, with a special meaning only when followed by `return`, so this should not break any existing code). This would be both a constraint on the operand (to make it eligible for TCE) and a requirement on the implementation (to apply the TCE). The standard wording would be something like the following:

If a `return` statement is preceded by `tail`, then evaluation of its operand will be a tail call, and the program is ill-formed if:

- the statement is within a *function-try-block*,
- the operand is not a function call expression whose *postfix-expression* has a function type,
- any live object with automatic storage duration within the scope of the enclosing function has a non-trivial destructor, or its address is taken or it is bound to a reference anywhere within the function body, or
- the function designated by the function call expression is not defined in the current translation unit, or has a return type that is not the same as the return type of the calling function, or has a *parameter-declaration-clause* that terminates with an ellipsis.

We believe that the above conditions are minimally sufficient to permit TCE in Clang, and probably in any other reasonable C++ implementation (of course, we particularly welcome implementer feedback on this point). Note that the generated code for a coroutine lambda can

easily ensure that all these conditions hold for its invocations of `operator[<-]`, except that it cannot guarantee that the operator is defined in the current translation unit. We will therefore specify that invocation of `operator[<-]` by a coroutine lambda is always a tail call *if* the selected overload is defined in the current translation unit.

**Alternative:** we could loosen the above rules somewhat to permit taking addresses of and forming references to local variables, but specify that the lifetime of local variables ends when the tail call begins (since we forbid nontrivial destructors, the effect of this is just that it's UB to access them after that point). However, that would make this construct less safe, since changing `return` to `tail return` could break code in ways that can't be detected at compile time.

**Alternative:** we could achieve the same behavior via an attribute, e.g. `[[tail_call]]`. This would be more conceptually lightweight than a new contextual keyword, correctly signalling to programmers that they can disregard this feature unless they have a specific need for it. However, an attribute might not allow us to normatively mandate TCE, which we believe is necessary.

**Alternative:** rather than allow users to force TCE, we could make it inherent in the API for unwrap expressions. Specifically, we could allow `operator[<-]` to return either the return type of the coroutine, or a nullary callback (with the same return type as `operator[<-]`). The unwrap expression which invokes `operator[<-]` would then apply a "trampoline" technique, repeatedly checking if the result is a callback, and if so invoking it to obtain a new result, until it obtains a final return value. However, this would substantially complicate the `operator[<-]` API, and would not have the benefit of allowing TCE in other contexts.

## constexpr

We have not worked through this issue in detail, but we see no obstacles to allowing coroutines to be `constexpr` (and uses of them to be core constant expressions) on the same terms as ordinary functions. The sample implementation given above cannot be `constexpr` because of its use of `reinterpret_cast`, but that is only as an expository way of depicting the compiler's management of the stack frame, which we know it can do in `constexpr` code because it already does.

## Alternative: Patching the TS

We believe the design presented above addresses all of our major concerns with the Coroutines TS. However, we expect that many committee members will consider this change too extensive to make in the C++20 timeframe (and we don't necessarily disagree). If WG21 is committed to shipping Coroutines as part of C++20, it should still be possible to address some of our concerns.

We could add first-class syntactic support for non-asynchronous use cases by replacing the `co_await` keyword with an operator token such as `[<-]`. After C++20, we could still introduce such a token as a synonym for `co_await`, although of course we could no longer remove `co_await`.

We could make the coroutine kind locally explicit via some form of introducer syntax. As a straw man example:

```
auto OpenFile(const string& filename) using future_coroutine<File> {  
    ...  
}
```

This would enable us to eliminate `coroutine_traits` (and hence eliminate the need for a shared global namespace of coroutine signatures), and also allow ordinary `return` in coroutines, although `co_return` would still be necessary in cases where e.g. the return value is not implicitly convertible to the return type. We could also add a capture group to the introducer syntax, to give explicit control of capture semantics:

```
auto OpenFile(const string& filename) using future_coroutine<File> [filename] {  
    ...  
}
```

## Allocation and performance

We believe the following is a consensus description of the Coroutines TS status quo:

- A conforming implementation is permitted to allocate every coroutine frame via `operator new`; neither [HALO](#) nor “suspend point simplification and elimination” is ever guaranteed to occur.
- No existing implementation reliably elides unnecessary allocations.
- Making allocation elision reliable will require ABI extensions that have not yet even been prototyped.
- It is not yet clear whether coroutine frame allocation elision will be reliable in the no-optimization modes of major compilers (after all, it is very explicitly an “optimization”).
- User code can unwittingly disable HALO, e.g. by allowing the coroutine object’s address to escape the coroutine, and it’s not yet clear how we’d teach users to avoid those hazards.
- RVO currently cannot be applied to coroutine returns.

Consequently, as one example, it is impossible to write a generator function that is guaranteed not to allocate, unless you can modify the function signature in order to trigger a custom `operator new` overload. We contend that in order for coroutines to be legitimately “zero overhead” for the generator use case, it must be possible to write a generator that is guaranteed not to allocate, if the corresponding non-generator-based code is guaranteed not to allocate (and uses only a bounded amount of stack).

Similarly, it is impossible to write a function that returns `expected<T,E>` and uses `co_await` for error propagation, but is guaranteed not to allocate.

We could address these problems through the following extensions:

- Extend the coroutine promise API with a static member `is_resumable`, which specifies whether coroutines that use that promise can be resumed. This will permit types such as `expected<T,E>` to opt out of support for resumption.
- Add wording to normatively require allocation elision when `is_resumable` is false, or when the conditions for HALO apply (e.g. the relevant operations are inlineable, and the coroutine object satisfies some specified set of conditions that imply that it does not escape).
- Specify that if a coroutine promise has an accessible static member `no_alloc`, and the program semantics permit the coroutine to be implicitly heap allocated, the program is ill-formed. This will permit programmers to ensure that a failure to satisfy the HALO conditions will manifest as a build failure, rather than a silent performance regression.
- Permit coroutines to be `constexpr`, and specify that when `resumable` is true, violations of the HALO conditions cause an expression to fail to be a core constant expression.
- Extend the coroutine promise API to expose the storage location where a return value should be constructed, in order to enable RVO in coroutines (we understand that Gor Nishanov is working on a specific proposal for this).

These all appear to be pure extensions, so they could be done post-C++20 if need be.

## API Complexity

We see no viable way to address the API complexity of the Coroutines TS via such incremental changes. Indeed, the changes we discuss will add yet more extension points, and we think it is likely that there will be a more or less perpetual drip of new extension points and new complexity, if we proceed with the TS design. The only way we see to fundamentally simplify coroutines is to give user code direct access to the primitive objects and operations that constitute the feature. So long as the primitives are hidden behind an abstraction boundary, it will remain necessary to poke holes in that abstraction in order to meet the needs of our diverse and highly performance-sensitive user community.

## Comparison

The following chart summarizes what we see as the key functional differences between the Coroutines TS status quo, the TS with incremental fixes, and our proposal:

	Coroutines TS	Incremental alternative	Core coroutines
Library	15:	18:	3 or 4:

<b>customization points</b>	<pre> await_transform operator co_await await_ready await_suspend await_resume yield_value return_value return_void promise_type get_return_object get_return_object_on_allocation_failure coroutine_traits initial_suspend final_suspend unhandled_exception </pre>	<pre> await_transform operator co_await await_ready await_suspend await_resume yield_value return_value return_void promise_type get_return_object get_return_object_on_allocation_failure coroutine_traits initial_suspend final_suspend unhandled_exception is_resumable no_alloc return_value_slot </pre>	<pre> operator[&lt;-] operator() return_type (Factory function)<sup>3</sup> </pre>
<b>Coroutine object representation</b>	Type-erased as <code>coroutine_handle</code>	Type-erased as <code>coroutine_handle</code>	Concrete object with anonymous type
<b>Coroutine allocation (normative)</b>	All coroutine objects are heap-allocated by default. This can be disabled by explicit collaboration between library and user code.	All coroutine objects are heap-allocated by default, but libraries can opt out. This constrains their usage to certain optimizable patterns, which seem to cover known common cases where allocation is unnecessary.  Implementations are normatively required to implement the necessary optimizations.	Coroutine objects are allocated by explicit code, just like all other objects. Allocation will normally be a hidden detail of the library.
<b>Coroutine allocation (QoI)</b>	Optimizers have demonstrated ability to elide coroutine allocations in many	Same as Coroutines TS.	Allocation elision applies equally to all kinds of objects, including

<sup>3</sup> Factory functions like `make_future` are not customization points in quite the same sense, since they have no special role in the language rules, and are directly exposed to users.

	common cases. Techniques sufficient to reliably elide allocation for specific types are on the drawing board. Unclear whether optimizations will apply in all build modes.		coroutines.
<b>(N)RVO in coroutines</b>	No	Yes	No NRVO if there's a suspend point between construction and return.
<b>Programmer control of capture</b>	No	Yes	Yes
<b>return in coroutines</b>	Forbidden	Allowed, but <code>co_return</code> is still needed in some cases.	Required ( <code>co_return</code> is unnecessary)
<b>User-facing syntax</b>	Keyword, concurrency-specific	Operator token, general-purpose	Operator token, general-purpose

## Conclusion

C++ is a language that enables programmers to build powerful and efficient abstractions by composing simple primitives that are efficiently supported by the platform. This is a defining property of C++, and a cornerstone of its success, so we should not abandon it (or even postpone it) without extremely compelling reasons.

The current design of the Coroutines TS is not consistent with that principle, because it does not provide simple, composable primitives, but only a complex abstraction that is tuned for a particular kind of use case. Shipping the current design as part of a C++ IS would be either an outright rejection of that principle or, at best, a wholly unjustified gamble that we'll be able to add the necessary primitives as a non-breaking extension, and still end up with a coherent design.

We believe that C++ can still be a vital language 50 years from now, and the language should be designed with that goal in mind. In 50 years nobody will even remember whether coroutines shipped in C++20 or C++23, but if we lock ourselves into a coroutines design that lacks such an essential ingredient of C++'s success, the consequences could easily last that long.

We have shown that a revised design that accords with that principle is well within reach, and that the resulting facility will be simpler, more general, and more efficient. We therefore urge the committee not to merge the Coroutines TS into the IS in its current form, and instead to allow sufficient time for this design to be fleshed out and validated.

## Acknowledgements

Many thanks to Richard Smith, Gor Nishanov, Roman Perepelitsa, Jeffrey Yasskin, Bryce Lebach, Michael Spencer, Davide Italiano and Gabriel Kerneis for their valuable design discussions and feedback.

# Appendix: Examples

Caveat: unless otherwise indicated, these examples are completely untested.

## Futures

The following is a (very) rough implementation of a future library that supports coroutines. All types other than `promise` and `future` are hidden implementation details. This implementation leaks all shared states, in order to avoid a lot of distracting reference-counting machinery:

```
// Interface of all future shared states. This API should be sufficient
// to implement future<T>.
template <typename T>
class future_shared_state {
public:
    virtual bool is_ready() const = 0;
    virtual T& get() const = 0;

    // Causes the result of this shared state to be fed into `continuation`.
    // `is_ready` and `get` cannot be called after this.
    virtual void fuse_to(promise<T> continuation) = 0;

    virtual ~future_shared_state() = 0;
};

// Interface of all promises.
template <typename T>
class promise_interface {
public:
    virtual void set_value(const T& value) = 0;
    virtual void set_exception(std::exception_ptr ptr) = 0;
    virtual ~promise_shared_state() = 0;
}

// Tag type representing a shared state that is not yet ready.
struct not_ready{};

// Tag type representing a shared state whose result has been computed, and
// immediately passed to a continuation.
struct consumed{};

// A shared state implementation for ordinary promise/future patterns.
template <typename T>
class concrete_shared_state : public promise_interface<T>, future_shared_state<T> {
```

```

variant<not_ready, T, std::exception_ptr, consumed> state_;
promise<T> continuation_;

std::mutex mu_;
std::condition_variable done_;

public:
bool is_ready() const override {
    lock_guard guard(mu_);
    assert(!holds_alternative<consumed>(state_));
    return !holds_alternative<not_ready>(state_);
}

T& get() const override {
    lock_guard guard(mu_);
    done_.wait(guard, [&] { return !holds_alternative<not_ready>(state_); });
    return std::visit(overloaded(
        [] (not_ready) -> T& { std::abort(); },
        [] (T& t) { return t; },
        [] (std::exception_ptr ptr) -> T& { std::rethrow_exception(ptr); },
        [] (consumed) -> T& { std::abort(); }));
}

void fuse_to(promise<T> continuation) {
    bool already_done = true;
    {
        lock_guard guard(mu_);
        assert(!holds_alternative<consumed>(state_));
        if (holds_alternative<not_ready>(state_)) {
            assert(!continuation_);
            continuation_ = std::move(continuation);
            already_done = false;
        }
    }

    if (already_done) {
        std::visit(overloaded(
            [&] (T& t) { continuation.set_value(t); },
            [&] (std::exception_ptr ptr) { continuation.set_exception(ptr); },
            [&] (auto&) { assert(false); })),
            state_);
        state_.emplace<consumed>();
    }
}

void set_value(const T& value) override {

```

```

promise<T> continuation;
{
    lock_guard guard(mu_);
    if (!continuation_) {
        state_.emplace<T>(value);
        done_.notify_all();
        return;
    }
    continuation = std::move(continuation_);
    state_.emplace<consumed>();
}
continuation->set_value(value);
}

void set_exception(std::exception_ptr ptr) override {
    promise<T> continuation;
    {
        lock_guard guard(mu_);
        if (!continuation_) {
            state_.emplace<std::exception_ptr>(ptr);
            done_.notify_all();
            return;
        }
        continuation = std::move(continuation_);
        state_.emplace<consumed>();
    }
    continuation->set_exception(ptr);
}
};

// A shared state co-allocated with a coroutine. Does not implement
// promise_interface, because the value is determined by running the coroutine.
template <typename T, typename Coroutine>
class coroutine_shared_state : public future_shared_state<T> {
    concrete_shared_state<T> shared_state_;
    Coroutine coroutine_;

public:

    // Enable this object to act as a final return functor.
    using return_type = void;
    template <typename U>
    void operator()(const U& u) {
        set_value(u);
    }

    coroutine_shared_state([] -> Coroutine coroutine)

```

```

    : coroutine_(coroutine()) {
    // Begin execution of the coroutine, and return the first time it
    // blocks.
    try {
        coroutine_(*this);
    } catch (...) {
        shared_state_.set_exception(std::current_exception());
    }
}

bool is_ready() const override { return shared_state_.is_ready(); }
void set_continuation(promise<T> continuation) override {
    shared_state_.set_continuation(continuation);
}

T& get() const override {
    return shared_state_.get();
}

template <typename U, typename SuspendPoint>
promise<U> make_resume_promise(SuspendPoint suspend_point) {
    struct resume_promise : public promise_interface<U> {
        coroutine_shared_state* parent;
        SuspendPoint suspend_point;

        void set_value(const T& value) override {
            try {
                parent_>coroutine_(value, suspend_point, *parent_);
            } catch (...) {
                parent_>set_exception(std::current_exception());
            }
        }
        void set_exception(std::exception_ptr ptr) override {
            parent_>coroutine_.raise(ptr, suspend_point, *parent_);
        }
    };
    return promise<U>(new resume_promise{this, suspend_point});
}

};

template <typename T>
class promise {
    // Invariant: if two promise objects have equal shared_state_ values, they are
    // both null.
    promise_interface<T>* shared_state_;

public:

```

```

promise(promise_interface<T>* shared_state)
    : shared_state_(shared_state) {}

promise(promise&& other)
    : shared_state_(other.shared_state_) {
    other.shared_state_ = nullptr;
}
promise& operator=(promise&& rhs) {
    shared_state_ = rhs.shared_state_;
    rhs.shared_state_ = nullptr;
}

explicit operator bool() { return shared_state_ != nullptr; }

void set_value(const T& value) {
    shared_state_>set_value(value);
}

void set_exception(std::exception_ptr ptr) {
    shared_state_>set_exception(ptr);
}
};

template <typename T>
class future {
    // Invariant: if two future objects have equal state_ values, they are both null
    future_shared_state<T>* state_;

    template <typename Coroutine>
    friend future<T> make_future<T, Coroutine>([] -> Coroutine coroutine);

    // Public API left as exercise for reader
};

template <typename T, typename Coroutine>
future<T> make_future([] -> Coroutine coroutine) {
    auto* state = new coroutine_shared_state<T, Coroutine>>(coroutine());
    state.run();
    return future<T>(state);
}

template <typename T, typename U, typename Coroutine, typename SuspendPoint,
        typename ReturnCallback>
void future<T>::operator[<-](
    Coroutine& coroutine, SuspendPoint suspend_point,
    coroutine_shared_state<U, Coroutine>& outer_state) [<-] T && {
    state_>set_continuation(outer_state.make_resume_promise<T>(suspend_point));
}

```

```
}
```

A typical usage, as shown earlier, could look like:

```
auto count_bytes(Connection& connection) => make_future<int>([&connection] do {
    int bytes_read = 0;
    vector<char> buffer(1024);
    while (!connection.done()) {
        bytes_read += [<-]connection.Read(buffer.data(), buffer.size());
    }
    return bytes_read;
});
```

## Simple generator

This example prints the contents of a binary tree in order, using a generator:

```
struct BstNode {
    BstNode* left, right;
    string value;
};

auto Traverse(BstNode<int>* node) => generator<string>([&node] do {
    if (node == nullptr) {
        return;
    }
    [<-] Traverse(node->left);
    [<-] std::yield(node->value);
    [<-] Traverse(node->right);
})

void PrintBst(BstNode* root) {
    generator<string> g = Traverse(root);
    while (g) {
        cout << *g << endl;
        g.next();
    }
}
```

And here's the implementation that supports it:

```
namespace std {
    // yield_handle represents the result of a `yield` call. It has no semantics
    // of its own; semantics are provided by the operator[<-] overloads for specific
    // generators. Thus, all generators can use the same `yield` function.
    template <typename T>
```

```

struct yield_handle {
    T& value;
};

template <typename T>
yield_handle<T> yield(T& value) {
    return {value};
}

template <typename T>
yield_handle<const T> yield(const T& value) {
    return {value};
}
} // namespace std

// The current state of a generator<T,P>. This is a hidden implementation
// detail, but it must be a namespace-scope template in order to facilitate
// deduction of T and P.
template <typename T, typename P>
struct generator_state {
    // The code to execute to resume this generator. Null if this generator
    // is done.
    std::function<state(P&)> continuation = nullptr;

    // Pointer to the currently yielded value. Null if this generator is done.
    T* value = nullptr;
}

// generator<T, P> represents a bidirectional generator, i.e. that not only
// yields values of type T, but takes arguments of type P (which become values
// of the yield expression). Yielded values are accessed by dereferencing,
// and the generator is advanced to the next yielded value by calling next().
// Like an iterator, a generator has a special past-the-end state, signifying
// the end of the generated sequence, which cannot be dereferenced or advanced.
//
// The generator<T, void> specialization (which represents a traditional
// unidirectional generator) is omitted for brevity; the differences are
// mostly obvious, but note that it could easily implement MoveIterator
// (see P0902R0).
template <typename T, typename P = void>
class generator {
    generator_state<T,P> state_;

    // Manages lifetime of the coroutine lambda. Is not accessed otherwise.
    std::unique_ptr<void, void(*)(void*)> coroutine_;

public:

```

```

friend void swap(generator& lhs, generator& rhs) {
    using std::swap;
    swap(lhs.state_, rhs.state_);
    swap(lhs.coroutine_, rhs.coroutine_);
}

// Move only
generator(generator&& rhs) { swap(*this, rhs); }
generator& operator=(generator&& rhs) { swap(*this, rhs); }

// Constructs a generator which exposes the values yielded by lazy_coroutine().
template <typename Coroutine>
generator([] -> Coroutine lazy_coroutine) {
    unique_ptr<Coroutine, void(*)(void*)> coroutine(
        new Coroutine(lazy_coroutine()),
        +[] (void* ptr) { delete static_cast<Coroutine*>(ptr); });
    state_ = (*coroutine)(implicit_convert<generator_state<T,P>>{});
    coroutine_ = std::move(coroutine);
}

// Returns whether the generator is dereferenceable. False indicates
// the end of the generated sequence.
explicit operator bool() const { return state_.continuation != nullptr; }

// Accessors for the currently yielded value. static_cast<bool>(*this) must
// be true. Valid only until the following `next()` call.
T& operator*() { return *state_.value; }
T* operator->() { return state_.value; }

// Advance to the next yielded value. static_cast<bool>(*this) must be true.
void next(P& p) {
    state_ = state_.continuation(p);
}
};

// operator[<-] on a generator object behaves like python's `yield from`: next()
// operations on the outer generator are delegated to the inner generator
// until it is done, and then the outer generator's coroutine is resumed.
// Consequently, it does not pass a value when resuming the coroutine, even for
// bidirectional generators.
template <typename T, typename U, typename P, typename Q, typename Coroutine>
generator_state<T, P> operator[<-] (
    generator<U, Q>&& inner_generator, Coroutine& outer_coroutine,
    size_t suspend_point, implicit_convert<generator_state<T, P>>&) [<-] void {
    tail return yield_from_impl<T,U,P,Q,Coroutine>(
        std::move(inner_generator), outer_coroutine, suspend_point);
}

```

```

// The implementation is factored out as a helper function, so that it can
// call itself recursively (`<-` can only be used inside a coroutine, and
// explicit invocation of `operator<-`() is disallowed).
template <typename T, typename U, typename P, typename Q, typename Coroutine>
generator_state<T, P> yield_from_impl(
    generator<U, Q>&& inner_generator, Coroutine& outer_coroutine,
    size_t suspend_point) {
    if (inner_generator) {
        return { [&inner_generator, &outer_coroutine, suspend_point] (P& p) {
            inner_generator.next(p);
            tail return yield_from_impl<T,U,P,Q,Coroutine>(
                std::move(inner_generator), outer_coroutine, suspend_point);
        },
            &*inner_generator};
    } else {
        tail return outer_coroutine(
            suspend_point, implicit_convert<generator_state<T, P>>{});
    }
}

// This overload defines the semantics of yielding from a generator<T,P>
template <typename T, typename P, typename Coroutine>
generator_state<T, P> operator<[->(
    yield_handle<T> handle, Coroutine& coroutine, size_t suspend_point,
    implicit_convert<generator_state<T,P>>& final_return) [-> P& {
    return { [&coroutine, suspend_point, return_type] (P& p) {
        return coroutine(p, suspend_point, final_return);
    }, &handle.value};
}

```

## Zero-allocation generator

The above generator is comparable to generators as proposed by the Coroutines TS; in particular, it allocates every coroutine state on the heap, which is extremely inefficient in many cases. The following example shows a generator that always stores its state on the stack, which isn't possible with the Coroutines TS (without changing the signatures of generator functions). As a consequence of storing its state on the stack, generator functions defined this way cannot recurse (i.e. the maximum generator stack depth must be statically known).

It should be possible to use similar techniques to define a generator library that supports recursion by using a side stack (i.e. at most one more allocation than the corresponding non-generator-based recursive code), but the API design of the side stack abstraction raises issues beyond the scope of this paper.

First, a usage example:

```
// Returns a generator whose output consists of the concatenated
// outputs of each generator produced by `generators`.
template <typename T, typename P>
auto flatten(stack_generator_base<stack_generator_base<T,P>>&& generators)
    => stack_generator<T,P>([&] do {
    while (generators) {
        [<-]&*generators;
        generators.next();
    }
}

// Returns a generator that iterates over the given range.
template <typename Range>
auto traverser(const Range& range)
    => stack_generator<decltype(*begin(range)), void>([&] do {
    for (auto& element: range) {
        [<-] std::yield(element);
    }
}

// Returns a generator that yields `f(x)`, for each `x` yielded by `g`.
template <typename T, typename F>
auto transform_generator(stack_generator_base<T>&& g, F f)
    => stack_generator<decltype(f(*g)), void>([&g, f] do {
    while (g) {
        [<-] std::yield(f(*g));
        g.next();
    }
}

// Toy example: turn a nested vector into nested generators, and then
// flatten them
void f(const std::vector<std::vector<int>>& vectors) {
    stack_generator<int> gen = flatten(transform_generator(
        traverser(vectors),
        [] (const std::vector<int>& vec) { return traverser(vec); }));

    while (gen) {
        // Do stuff with *gen
        gen.next();
    }
}
```

And the underlying implementation:

```

// The internal state of a stack_generator<T,P,Coroutine>
template <typename T, typename P>
struct stack_generator_state {
    // Pointer to the currently yielded value
    T* value = nullptr;

    // Suspend point at which to resume the coroutine
    size_t suspend_point = SIZE_MAX;

    // The generator we have recursed into, if any
    stack_generator_base<T, P>* nested_generator = nullptr;
};

// Base class of all stack_generators that take P and yield T.
// Allows us to type-erase the coroutine.
template <typename T, typename P>
class stack_generator_base {
public:
    void next(P& p) {
        state_.value = next_impl(p);
    }

    operator bool() const {
        return state_.suspend_point != SIZE_MAX || state_.nested_generator != nullptr;
    }
    T& operator*() { return *state_.value; }
    T* operator->() { return state_.value; }

private:
    template <typename T2, typename P2>
    friend class stack_generator_base<T2, P2>;

    // Resumes execution of the generator, and returns the new state
    virtual stack_generator_state<T,P> resume(P& p, size_t suspend_point) = 0;

    T* next_impl(P& p) {
        if (state_.nested_generator != nullptr) {
            T* value = state_.nested_generator.next_impl(p);
            if (value != nullptr) {
                return value;
            } else {
                state_.nested_generator = nullptr;
            }
        }
        assert(state_.nested_generator == nullptr);

        if (state_.suspend_point == SIZE_MAX) {

```

```

        return nullptr;
    }
    state_ = resume(p, state_.suspend_point);
    return state_.value;
}

stack_generator_state<T,P> state_;
};

template <typename T, typename P = void, typename Coroutine>
class stack_generator : public stack_generator_base<T,P> {
public:
    stack_generator([] -> Coroutine coroutine)
        : coroutine_(coroutine()) {}

    // Make stack_generator usable as a final return functor.
    using return_type = stack_generator_state<T,P>;
    stack_generator_state<T,P> operator>()() { return {}; }

private:
    Coroutine coroutine_;

    stack_generator_state<T,P> resume(P& p, size_t suspend_point) override {
        return coroutine_(p, suspend_point, *this);
    }
};

template <typename T, typename P, typename Coroutine>
stack_generator_state<T, P> operator[<-](
    std::yield_handle<T> handle, Coroutine&, size_t suspend_point,
    stack_generator<T,P>&) [<-] P& {
    return {handle.value, suspend_point, nullptr};
}

template <typename T, typename U, typename P, typename Q,
        typename OuterCoro, typename InnerCoro>
stack_generator_state<T, P> stack_generator<U, Q, InnerCoro>::operator[<-](
    OuterCoro&, size_t suspend_point, stack_generator<T,P>&)
    [<-] void {
    return {value_, suspend_point, this};
}

```