

Document number: P0973R0

Date: 2018-03-23

Reply To: Geoff Romer (gromer@google.com), James Dennett (jdennett@google.com)

Audience: WG21 Evolution Working Group

Coroutines TS Use Cases and Design Issues

[Use cases](#)

[Issues identified](#)

[Implicit allocation violates the zero-overhead principle](#)

[Const reference parameters are dangerous](#)

[Banning return is user-hostile, and makes migration difficult](#)

[The library bindings are massive and yet inflexible](#)

[The name `co_await` privileges a single use case](#)

[`co_await` nests awkwardly](#)

[`constexpr` is not supported](#)

[Impact on use cases](#)

[Concurrency](#)

[Error propagation](#)

[Generators/Ranges](#)

[Conclusion](#)

Now that coroutines are available in a TS, and supported by some implementations, we have been able to evaluate in detail the possibility of deploying them in our primary codebase. On the basis of that evaluation, we believe that coroutines have the potential to solve several problems for C++ programmers at Google, and moreover those problems are likely serious enough, and the potential solution good enough, to justify the the risk of adopting coroutines prior to full standardization. However, we have identified a number of issues with the Coroutines TS in its current form, which we believe will completely prevent us from using them for one of our major use cases, and seriously hamper our use for the other.

We are already actively developing a set of proposed revisions to the Coroutines TS to address these issues, which we intend to present in Rapperswil. However, we believe any fix will require substantial non-backwards-compatible changes to the Coroutines TS design, and so should be done before coroutines are published in a C++ IS. We are only publishing these concerns in advance of our proposed fixes because we believe the committee should be aware of them when evaluating [P0912](#)'s proposal to merge the Coroutines TS into the C++ IS.

Use cases

One of our primary use cases for coroutines is the one that obviously motivates the Coroutines TS, namely fine-grained asynchrony and concurrency. Generators are a second; they are not a major concern for us at present, but we believe that they could be a critical use case in the future, because we see preliminary indications that generator-based programming techniques may give us important performance improvements when combined with the Ranges TS.

Presumably this audience needs no further introduction to those use cases, but a third use case has not been discussed as extensively: coroutines enable concise and convenient propagation of errors via return values. This is useful for cases where errors are common rather than exceptional, the error path is performance-sensitive, and/or exceptions are not permitted (which over 50% of respondents to the C++ Developer Survey have to deal with in at least some contexts). Functions that return `T` on success can report errors by returning a type such as `expected<T>` (as proposed by [P0323](#); we omit the error space parameter for brevity), but in C++17 this requires an onerous degree of error handling boilerplate:

```
expected<string> GetString(int index);
expected<int> SumOfSizes() {
    int result = 0;
    for (int i = 0; i < kMaxIndex; ++i) {
        expected<string> tmp = GetString(i);
        if (!tmp.has_value()) {
            return unexpected(tmp.error());
        }
        result += tmp->size();
    }
    return result;
}
```

Changes in the API design of `expected<T>` could mitigate some of this boilerplate, but not the fundamental necessity of storing an `expected<T>` function result in a named temporary, checking it for error and conditionally returning, and then using the stored value. In practice, we have been unable to prevent users from using macros to abbreviate this process:

```
expected<int> SumOfSizes() {
    int result = 0;
    for (int i = 0; i < kMaxIndex; ++i) {
        ASSIGN_OR_RETURN(expected<string> tmp, GetString(i));
        result += tmp->size();
    }
    return result;
}
```

Coroutines have the potential to provide a far more satisfactory solution; with suitable bindings for `expected<T>`, this could be rewritten as:

```
expected<int> SumOfSizes() {
    int result = 0;
    for (int i = 0; i < kMaxIndex; ++i) {
        result += (co_await GetString(i)).size();
    }
    co_return result;
}
```

Note also that there is already a [proposal](#) to introduce a separate syntax for this use case. We think that it would be much better for C++ to have a single syntax that is expressive enough to support all coroutine-like use cases, rather than inventing a new syntax for each.

Issues identified

This section lists the major issues we've identified with the Coroutines TS, in non-increasing order of importance.

Implicit allocation violates the zero-overhead principle

The Coroutines TS specifies that a coroutine may allocate memory on the heap, in order to store the execution state of the suspended coroutine. In cases such as `expected`-based error handling, this is unnecessary because the coroutine will never be resumed after suspension (because suspension occurs only when an error is encountered). This unnecessary allocation is a serious problem, because it means that modifying a function to use `co_await` instead of e.g. `ASSIGN_OR_RETURN` can cause that function to allocate memory, a potentially major performance regression.

The authors of the Coroutines TS argue that the allocation can be optimized away when it is not needed, but there are several problems with this argument:

- It is not yet clear that that this optimization is feasible in general. For example, the Clang implementation of the Coroutines TS can perform this optimization only when the coroutine is in the same translation unit as the caller (among other conditions), and Richard Smith says “I do not have any confidence this optimization would ever become reliable in the way that, say, certain forms of NRVO are.”
- Even if implementations are eventually able to apply this optimization reliably, C++ programmers are unlikely to trust it to do so. As a point of comparison, consider the case of copy elision: this optimization has been universal in C++ compilers for over a decade, and yet C++ programmers still routinely avoid returning large objects by value because they're concerned about performance.

- This approach is at odds with the design philosophy of C++. One of C++'s most distinctive advantages over other languages is that it enables programmers to define and use powerful abstractions, while still giving them fairly precise visibility into and control over performance when necessary. The Coroutines TS's approach to allocation compromises that advantage because it fails to give programmers an accurate mental model of their code's performance.

In addition to giving users no visibility or control over *whether* that allocation takes place, the Coroutines TS does not provide enough control over *how* that allocation takes place. Allocation of the coroutine frame can be controlled only by overloading `operator new`, an awkward and obscure extension point with too narrow an API to express all the ways that coroutine type authors may want to manage coroutine state. For example, the creator of a generator type may want to allocate nested generators as a stack in a single contiguous region of memory, so that generator performance matches the performance of ordinary function calls. The Coroutines TS does not give the author the tools to do this, because allocation is determined solely by the parameters of the coroutine function, and the coroutine type author does not control its users' parameter types.

The problems with the Coroutines TS's current approach to allocation are comparable to the problems we would have if we had tried to specify `std::vector` as C++'s fundamental representation for arrays: this would naively imply that creating an array always results in a heap allocation, and many of us would not be satisfied by the argument that this could be optimized away when the compiler determines that the size is fixed.

Const reference parameters are dangerous

Consider the following coroutine:

```
future<string> Concat(const string& prefix, future<string> suffix) {
    co_return (prefix + co_await suffix);
}
```

Despite its innocuous appearance, this code has a serious bug: if `Concat` is invoked with a temporary `prefix` argument (e.g. `Concat("foo", GetSuffix())`), `prefix` may be a dangling reference by the time it is accessed.

To some extent, this problem is inherent in the very concept of allowing functions (or function-like entities) to suspend execution, and then resume execution in a context that's independent of the original callsite: without profound changes to the C++ object lifetime rules, the temporaries created at the callsite will necessarily no longer exist when execution resumes. However, the fact that this problem is concealed behind a familiar and innocuous-seeming syntax is not inherent. For example, a very similar problem arises in connection with lambdas,

where a reference that's captured when a lambda is created may be dangling by the time the reference is used. In that case, C++ opted to use a different syntax for variable capture than for argument passing, which has enabled us to teach users different rules for capturing by reference (use with caution, pay attention to lifetime issues) vs. passing by reference (use freely).

Note that banning reference parameters, either globally or for particular coroutine types, is not an adequate solution to this problem, because it would disallow many important use cases. We want to make this functionality available to all coroutines, but in a form that does not give users a false sense of security.

Banning return is user-hostile, and makes migration difficult

The Coroutines TS currently forbids coroutines from using `return` statements; they must instead use `co_return`. This is motivated by the semantic differences between ordinary and coroutine return (particularly the indirection through the `return_value/return_void` extension points), as well as to simplify single-pass implementations. However, in practice the differences between `return` and `co_return` are minor for well-behaved coroutines, and the keyword difference does not make the semantic differences clear, so we are concerned that this will come across to users as arbitrary or confusing. Perhaps more problematically, this requirement makes it substantially more difficult to migrate existing macro-based error handling code to use `co_await` instead, because we must introduce `co_await` and eliminate `return` in a single atomic step.

The library bindings are massive and yet inflexible

The Coroutines TS couples the core language with the library to a degree that's simply without precedent. Although nominally a core language feature, the Coroutines TS depends tightly on a library type `coroutine_handle`, and hooks into no fewer than 15 novel user-controlled extension points¹ to customize its behavior (that's not counting the pre-existing `operator new`,

¹ `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`

which is likely to be pressed into service for new and strange purposes, since it's the only leverage the user has over that critical allocation).

This complex coupling makes coroutines more difficult to understand. It also, perversely, reduces library flexibility: the large API surface that user-provided code must expose to the core language makes it much harder for programmers to use coroutines in any way that the language designers did not anticipate, and we are concerned that the implicit core/library coupling inside `coroutine_handle` may mean that some kinds of implementation changes require NxM coordination between vendors of compilers and standard libraries (however, implementation experience so far has not borne out this concern).

More fundamentally, this extensive library coupling points to a possible design problem: the boundary between the language feature and the library may have been drawn in the wrong place. In effect, the Coroutines TS specifies the behavior of `co_await` and `co_yield` as elaborate multi-step algorithms, which call out to user-provided extension points at each step. This allows the user to customize each step, but not the algorithm as a whole. A design in which the entire algorithm was delegated to library code, leaving the core language responsible only for the fundamental operation of reifying the function state as a continuation, would be both simpler and more flexible.

Furthermore, the library extension points are to a large extent keyed off of the function signature (by way of the `coroutine_traits` template), so users are not able to effectively control the behavior of a single function, only of the function *and all other functions* that have the same parameter and return type. This effectively disallows per-function customization, while leaving an opening for unwary or unscrupulous users to *attempt* per-function customization in a highly hazardous way.

The name `co_await` privileges a single use case

At risk of stating the obvious, the `co_await` keyword privileges the concurrency use case to the exclusion of all others: it is unnatural at best to “await” an `expected` object or a parse tree. For example, it is significant that for generators, evidently the Coroutines TS's second-priority use case after asynchrony, it introduces a new purpose-specific syntax (and new library bindings) rather than reuse `co_await`. At first glance, generators seem conceptually distinct from both the future and error-handling use cases, since the latter focus on different kinds of “unwrapping” operations, whereas generators if anything seem to emphasize wrapping. Nonetheless, it is possible to implement generators quite naturally within the framework of `co_await`, using a syntax such as `co_await std::yield(x)`, although this might be less compatible with future extensions to enable return type deduction in coroutines.

If not even the Coroutines TS itself reuses `co_await` for use cases that the TS was explicitly designed for, that calls into question whether users will be willing to reuse `co_await` for use

cases that the TS didn't consider. See also [P0779](#)'s proposal of a separate syntax for the `expected` use case.

`co_await` chains awkwardly

If a user wishes to apply a postfix operator to the result of a `co_await` expression, they will generally have to wrap the expression in parentheses, or the operator will bind to `co_await`'s argument rather than its result. Particularly in the case of the `.` operator, this risks leading to a fairly parenthesis-intensive coding style:

```
int size = (co_await GetString(some_complicated_expression)).size();
```

This is a readability problem, because readers must count parentheses in order to read the code correctly.

Note that this problem is exactly analogous to the one solved by the `->` operator, with `co_await` in place of the `*` operator.

`constexpr` is not supported

The Coroutines TS forbids coroutines from being `constexpr`, which limits their utility somewhat. This is presumably because supporting coroutines in their current form at compile time would pose some formidable challenges, particularly since coroutine execution is specified to always allocate, and allocation is forbidden in `constexpr` code. A design built on simpler primitives should be able to support `constexpr` in a natural way.

Impact on use cases

Concurrency

Unsurprisingly, the Coroutines TS is reasonably well-suited to concurrency, its primary use case. The problem of implicit allocation is reduced, because allocation will almost always be necessary in concurrent use cases anyway. `co_return` will be an annoyance for users, but not an obstacle to migration, because our codebase's existing concurrency abstractions are too different from `std::future` for automated migration to be feasible in the first place. The name of `co_await` is perfectly suited to this use case (except for the abominable `co_` prefix, of course).

The biggest problem in the concurrency context will be the issue of dangling reference parameters. This will require extensive educational work (and tooling work, if possible) to make

users aware of this pitfall, and will nonetheless probably lead to many bugs (as if concurrent programming wasn't bug-prone enough to begin with).

That being said, this problem seems impossible to fully solve except by disallowing reference parameters to coroutines, which would probably be too restrictive (and which we could implement as a build or clang-tidy warning on top of the Coroutines TS, if we thought that was desirable). We think a better design for coroutines could at best mitigate this problem, by making the hazards more clear to users, leveraging our existing educational work on the hazards of lambda reference captures.

In short, despite a number of shortcomings, the Coroutines TS is probably usable in its current form as a solution for the problem of writing efficient, portable asynchronous code.

Error propagation

In contrast, the Coroutines TS appears to be unusable in its current form to solve the problem of simplifying error propagation and eliminating error-propagation macros. The ban on `return` statements in coroutines (and the symmetric ban on `co_return` in non-coroutines) makes it impossible to incrementally migrate usages of e.g. a particular macro to use `co_await`. Instead, each whole function must be migrated in a single atomic step. Thus, we cannot perform the migration by rewriting the macros to use `co_await`, nor can we perform it by developing migration tools for the macros one at a time. Instead, we must develop a single migration tool which can handle all `return` statements in a function (including those hidden behind any macros), a much more complex proposition.

Even if that problem were solved, we could not perform such a migration if there was any significant chance of a performance regression, and due to the implicit allocation specified by the Coroutines TS, we have no way of ruling that out.

Generators

As noted above, there are some preliminary indications that the upcoming Ranges TS (which is intended to become the preferred way to work with sequential data in C++) may require use of generators to attain optimal performance in at least some cases. However, the Coroutines TS's specification of implicit allocation could seriously degrade the performance of generators, and/or give them a reputation for poor performance that would discourage their use in performance-sensitive code. It's too early to assess this risk in much detail, except to say that the uncertainty is mostly on the downside.

Conclusion

The Coroutines TS in its current form is unusable for one of our critical use cases (error propagation), has serious safety concerns for the other (concurrency), and may be unsuited for a potentially major forward-looking use case (generators). It also suffers from a number of serious but not deal-breaking deficiencies that will make coroutines a perpetual source of friction for programmers.

Given the severity of these problems, and how deeply embedded they are in the design of the Coroutines TS, we do not believe the TS is ready to be moved into an International Standard. We strongly support making coroutines available to C++ programmers as quickly as possible, but we believe moving the current TS into C++20 would actually impede that goal. Consequently, we urge the committee not to adopt [P0912R0](#) until these issues are addressed, and we intend to follow up in Rapperswil with a proposal to address them.

Revision History

Since 2018-03-06 draft (as presented in Jacksonville):

- Corrected errors in usage of `expected<T,E>`.
- Clarified that Clang relies on the coroutine to be in the same translation unit, not necessarily inline, to elide coroutine frame allocation.
- Clarified that lack of control over allocation is an issue for coroutine type authors more than coroutine end-users, and illustrated with a more compelling example.
- Discussed option of banning reference parameters.
- Softened concerns about MxN coupling between compilers and standard libraries, based on implementer feedback.
- Called out potential benefit of `co_yield` for return type deduction.
- Rephrased to talk about `co_await`'s difficulty with *chaining*, rather than *nesting*.

Acknowledgements

Thanks to Gor Nishanov, Chandler Carruth, Richard Smith, J.F. Bastien, and many others, for feedback on previous drafts of this paper.