`return_value` & `return_void` Are Not Mutually Exclusive

Document Number: P3950R0
Date: 2025-12-21
Reply-to: Robert Leahy <rleahy@rleahy.ca>
Audience: EWG, SG1, SG17

Abstract

This paper proposes that the *function-body* of a function which is a coroutine be able to simultaneously contain statements both of the form `co_return;` and `co_return v;`.

Background

The standard specifies the effect of `co_return` statements in terms of equivalent statements within the context of a *replacement body* (§9.6.4 [dcl.fct.def.coroutine]). Like a regular `return` statement `co_return` statements have two distinct forms: Those that return `void` and those that return some value. These are specified as follows (§8.8.5 [stmt.return.coroutine]):

"If the operand is a braced-init-list *or an expression of non-*`void` *type,* `S` *is* `p.return_value(expr-or-braced-init-list)`*. [...] Otherwise,* `S` *is the* compound-statement `{ expression`$_{opt}$ `; p.return_void(); }`*."*

At first this appears to permit coroutines whose body contains `co_return` statements of void and of non-void provided the promise type admits corresponding invocations of `return_value` and `return_void`. Unfortunately the standard bans this by fiat (§9.6.4 [dcl.fct.def.coroutine]):

"If searches for the names `return_void` and `return_value` in the scope of the promise type each find any declarations, the program is ill-formed."

This restriction has been present in its current form since N4499 ([1] at §6.6.4):

"If the promise type defines both `return_value` and `return_void` member functions, the program is ill-formed."

A different form of this restriction was present in N4403 ([2] at §18.11.3):

"A promise type must contain at most one declaration of `set_result`."

These early coroutine proposals contained the notion of a coroutine's "eventual type." This concept was eventually dropped (there is no trace thereof in [3] or in the working draft).

There has previously been a paper by a different author with the same goal as this paper [4]. It had no consensus in Cologne in 2019, however the author of this paper feels there is new information [5][6].

Discussion

Implementing Promise Types

The restriction in the current working draft says (emphasis added):

*"If searches for the **names** [...] find any **declarations**, the program is ill-formed."*

Note the reference to "names" and "declarations." This restriction can only be implemented by the compiler. Regular C++ code cannot check for "names" or "declarations," it can only check for well-formed expressions. The above restriction does not require the expressions `promise.return_void()` or `promise.return_value(vs...)` (for some/any invented `vs`) to be (or not to be) well-formed, instead it says that if the names are found the program is ill-formed.

A consequence of the above is that promise types cannot be implemented using the following strategy:

```
template<typename ReturnType>
struct some-promise-type {
  void return_void() requires std::is_same_v<void, ReturnType> { /* ... */ }
  template<typename T = ReturnType>
    requires std::is_same_v<T, ReturnType>
  void return_value(T t) { /* ... */ }
  // ...
};
```

Because despite the fact expressions which invoke `return_void` and `return_value` will never be simultaneously well-formed the names are declared.

Coroutines With Heterogeneous Return Types

C++ functions return in exactly one way, and they return exactly one type. This is the case despite the irregularity of `void` [6] and the existence of `[[noreturn]]`. Importantly this means that there is no way, as a first-class feature of the language, for a C++ function to:

- Return in zero ways (e.g. always throw) (note that despite the existence of `[[noreturn]]` there is no generic way to inspect a function or invocable and ascertain that it will never yield a value)
- Return in multiple ways (e.g. `int` or `double`)
- Return multiple types (e.g. both `int` and `double`)

The latter two have library solutions, but at the language level those library solutions present as a single return modality with a single type (note that even where a function call expression is immediately bound to structured bindings the function call expression still yields a single value of a single type).

While the body of a C++ coroutine may syntactically resemble the body of a C++ function it is nothing of the sort. The standard makes this clear:

- The function body of a function which is a coroutine is rewritten so that it is no longer a function body, but instead a protocol by which the code interfaces with the promise (§9.6.4 [dcl.fct.def.coroutine])
- When a coroutine is invoked the value yielded thereby is not determined by the function body but instead by evaluating the `get_return_object` or `get_return_object_on_allocation_failure` nullary invocable member functions of the promise (ibid.)
- Allowing control to flow off the end of a coroutine is either undefined behavior, or equivalent to *`promise`*`.return_void()` (ibid.)
- The statements by which regular functions end their execution are disallowed in the function body of a coroutine (§8.8.5 [stmt.return.coroutine]) even if those statements are discarded (ibid.)

C++ is sufficiently powerful that the library may be used to fill in for missing language features. One instance of this is discussed above: Library features adding the ability for C++ functions to, in effect, return multiple values and in multiple ways. `std::execution`, which will ship in C++26, provides “the [library] implementation of an *`async-function`*” ([8] at §6).

The library implementation of a function provided by `std::execution`’s senders and receivers is breathtakingly more powerful than C++’s language-level functions, being:

- Fundamentally asynchronous through the decoupling of initiation and completion
- Capable of:
  - Abandoning forward progress via `std::execution::set_stopped`
  - Completing with errors beyond an exception throw
  - Expressing the concept of a function which:
    - Completes successfully with no values without the use of an irregular type
    - Does not complete successfully
    - Completes successfully in multiple ways (obviating the need for a separate sum type)

        - Completes successfully with multiple values (obviating the need for a separate product type)

The capabilities with respect to successful completion are expressed by an instantiation of `std::execution::completion_signatures` whose template arguments include, respectively:

- `std::execution::set_value_t()`
- No types of the form `std::execution::set_value_t(...)`
- Multiple types of the form `std::execution::set_value_t(...)`
- At least one type of the form `std::execution::set_value_t(T, U, ...)`

Were coroutines C++ functions this would leave us in an awkward place: "[A] Standard C++ model for asynchrony" (i.e. `std::execution`) which is library-based with capabilities wildly in excess of the corresponding language feature (i.e. coroutines).

Fortunately, as discussed above, coroutines are not C++ functions. They are a protocol for interacting with the promise. The shell of a C++ function which surrounds a coroutine exists only to yield a return object from the promise.

The promise is simply an instance of a C++ type. C++ types may have member function templates. Templates permit metaprogramming. Therefore promises can be authored which expose the power of the `std::execution` model:

- `co_return` statements need not accept an expression with the same type, or even with a common type, provided `return_value` can be invoked with the result of that expression, and therefore behind the scenes these can be plumbed through to different `std::execution::set_value` completion signatures
- The tuple-like protocol can be used to resolve individual values to `std::execution::set_value` completion signatures with multiple values

The above works. The author has implemented it.

Trying to accept `std::execution::set_value_t()` (i.e. successful completion with no values) alongside any other `std::execution::set_value_t(...)` form, on the other hand, does not work. Not for any conceptual reason, but simply because the standard bans it by fiat.

For further support of the above consider: The author of such a promise type can accept a value of a special tag type and map it to `std::execution::set_value(rcvr)` (the author has implemented this), but cannot simply write a `return_void` member function which maps to the same.

Conclusion

Disallowing `return_void` alongside `return_value` is fundamentally arbitrary, unnecessarily making `void` a special case. The method by which it is disallowed unnecessarily restricts the ways in which generic promise types can be implemented. Disallowing it either disadvantages coroutines vis-à-vis `std::execution` or necessitates library workarounds (e.g. the tag type approach discussed in the preceding section). Said restriction should for all the preceding reasons be removed.

Wording

[dcl.fct.def.coroutine]

~~If searches for the names `return_void` and `return_value` in the scope of the promise type each find any declarations, the program is ill-formed.~~

~~[*Note 2:* ~~If the expression *`promise`*`.return_void()` is ~~found~~well-formed, flowing off the end of a coroutine is equivalent to a `co_return` with no operand. Otherwise, flowing off the end of a coroutine results in undefined behavior.~~ — end note]~~

Acknowledgements

The author would like to thank Lewis Baker for permission to continue pursuing a solution to this problem.

References

[1] G. Nishanov et al. Draft wording for Coroutines (Revision 2) N4499
[2] G. Nishanov. Draft wording for Resumable Functions N4403
[3] Programming Languages — C++ Extensions for Coroutines N4680
[4] L. Baker. Allowing both `co_return;` and `co_return value;` in the same coroutine P1713R0
[5] M. Dominiak et al. `std::execution` P2300R10
[6] D. Kühl et al. Add a Coroutine Task Type P3552R3
[7] M. Calabrese. Regular Void P0146R1
[8] K. Shoop. async-object - aka async-RAII P2849R0