pmr::generator - Promise Types are not Values

Steve Downey (sdowney@gmail.com) (sdowney2@bloomberg.net)

Document #: P2787R0
Date: 2023-02-06
Project: Programming Language C++
Audience: LEWG, LWG

Abstract

The type returned from a coroutine is not a value semantic container. It nonetheless can satisfy the requirements for supporting pmr generators. It is useful to provide a pmr alias to make the requirement to use pmr allocators visible in the typesystem.

Contents

1 What is being proposed

Provide a template alias for std::generator that changes the default from an erased allocator to a pmr::allocator to facilitate enforcing business rules and semantic requirements in the typesytem.

2 Before / After Table

```
std::pmr::monotonic_buffer_resource mbr;
std::pmr::polymorphic_allocator<int> pa{&mbr};

std::generator<
    int,
    void,
    std::pmr::polymorphic_allocator<int>> g
      = pmr_requiring_coroutine(
          std::allocator_arg,
          pa);
```

```
std::pmr::monotonic_buffer_resource mbr;
std::pmr::polymorphic_allocator<int> pa{&mbr};

std::pmr::generator<int> g
  = pmr_requiring_coroutine(
      std::allocator_arg, pa
  );
```

3 PMR is not POCMA

Value semantic container types must be able to propagate their allocator and extend their move and move assignment to copy from one memory arena into another. They should ensure that the objects they contain are given the allocator they hold in order to ensure consistency of allocator scope.

Coroutine types do none of that. They do no allocation beyond the initial allocation of the coroutine frame and any housekeeping state used to manage deallocation. They, quite properly, do not advertise an `allocator_type` suggesting that they are `Allocator Aware`. They are not.

The `co_awaitable` returned from a coroutine has a handle to an uncopyable and immovable coroutine frame. The `coroutine_handle` has pointer semantics, a reference semantic type with value semantics. The promise type it is coupled with will often have smart pointer like semantics. However this all is significant during the creation of the the awaitable that controls the coroutine, and at its destruction. Allocation is not part of the operations modeled by the awaitable.

In particular the usage of the allocator that was passed into the coroutine that acts as the factory for the awaitable is not relevant and outside the span of control of the awaitable. That is to say it is not the concern of the generator if the string it produces uses the allocator.

The design intent of the pmr namespace is to make usage of pmr the default and to make that usage visible in the typesystem. It is not merely for allocator aware value types with an allocator template parameter that can be made to use pmr::allocator.

Anyone working with pmr allocators will eventually create the proposed alias, if just to avoid having to explain the second template parameter for generator. The standard should provide this to avoid the utterly needless creativity.

4 Generator is not a Container

The proposed `std::generator` is not a container. That means that Propagating On Container Move Assignment is a category error. It does not apply.

`std::generator` is a `range`, but ranges are not, necessarily, containers.

A function that returns a `std::generator` is either a factory co_routine function, or returning a `std::generator` that originated from such a factory. It is conceptually more like construction than function return.

5 Generators in Containers

`std::generator` does not advertise itself as an allocator aware type, so containers will not supply their allocator on copy or move to provide scoped behavior. This is appropriate as the frame, managed by the promise, and referenced by the coroutine handle, is immovable. In this sense a generator is as allocator aware as a T* or smart pointers produced by factories that use allocator objects.

6 Hiding Broken Behavior in Template Parameters is Bad

If a `pmr` alias for `std::generator` is inappropriate because the semantics are wrong, then the semantics are wrong without the alias. `std::generator` should be ill formed for the non-type erased form, not just for pmr::allocator but for any stateful allocator.

If something is ill formed, it should be ill formed no matter what syntax is used to express it.

If it is not ill formed, it is less appropriate for the standard library to allow but make difficult to express a type.

7 What About Task/Lazy

Technically this paper does not need to answer that. But we're trying to set policy. A coroutine task type will produce the same result for all observers. The storage for the potentially unrealized computation result should not be observable, and allocator propagation should not apply until the result value is read. Before that we are observing a singular pure value, immovable and uncopyable, barely existing.

The PMR model predates the modern value category types. Allocator propagation and scope is about value semantic types and does not apply to non-value types, in particular owning types with reference semantics.

We need not settle this now. Edge and corner cases are always up for question.

8 Base `namespace pmr` policy

A library type with a defaulted allocator template parameter, of any kind, should have a pmr alias that changes the default to be a pmr::allocator. This is not a change in design, although currently the pmr types are all regular value types with *allocator aware* semantics.

9 Wording

The proposed changes are relative to the current working draft [N4917].

```
namespace std {
  // coro.generator.class, class template generator
  template<class Ref, class V = void, class Allocator = void>
  class generator;

  namespace pmr {
    template<class R, class V = void>
    using generator = std::generator<R, V, polymorphic_allocator<R>>;
  }

}
```

References

[N4917] Thomas Köppe. N4917: Working draft, standard for programming language c++. https://wg21.link/n4917, 9 2022.