Revisiting allocator model for coroutine lazy/task/generator.

Document Number: **P 1681 R0** Date: 2019-06-11
Reply-to: Gor Nishanov (gorn@microsoft.com)
Audience: LEWG
Target: C++23

1 Introduction

Currently, proposed lazy<T> (formerly task<T>) coroutine type [P1056R1] uses an allocator customization model inspired by std::promise<T>.

task<T>/lazy<T>	std::promise<T>

`// Uses default allocator`
`// to allocate coroutine state`
`task<int> g(string s);`
 	
`// Uses provided allocator a.`
`template <class Allocator>`
`task<int> h(`
`  allocator_arg, Allocator a);`
 	
`// Uses provided allocator a.`
`template <class Allocator>`
`task<int> f(`
`  allocator_arg, Allocator a,`
`  int x, int y)`
`{`
`  ...`
`  co_await h({}, a);`
`  ...`
`}`
	`promise(); // Uses default allocator to allocate`
	`  // shared state.`
	
	`// Uses provided allocator a.`
	`template <class Allocator>`
	`promise(allocator_arg_t, const Allocator &a);`

This model is reasonable when coroutines are allocated with stateful allocators, but it leads to proliferation of needless boilerplate code when used with stateless allocators by forcing all coroutines to carry two extra parameters. This can be avoided if std::task/lazy would follow a std::vector<T, alloc> model that makes an allocator a part of the type.

Current Specification (P1056R1)	Improved (add allocator template argument)

`task<int> g(`
`  allocator_tag_t, MyAlloc,`
`  string s);`
 	
`task<int> f(`
`  allocator_tag_t, MyAlloc,`
`  int x, int y)`
 	
`{`
`  ...`
`  co_await g({},{}, “hello”);`
`  ...`
`}`
	`namespace my {`
	`  template <class T>`
	`  using task<T> = std::task<T, MyAlloc>;`
	`}`
	
	`my::task<int> g(string s);`
	
	`my::task<int> f(int x, int y){`
	`  ...`
	`  co_await g(“hello”);`
	`  ...`
	`}`

We would like to request LEWG guidance on the best way of addressing this deficiency in lazy<T>/task<T> type.

The approach chosen will impact upcoming standard coroutine types like generator<T>, async_range<T> and is likely to be adopted by non-standard coroutine libraries as well.

Let's consider a few alternatives.

2 Simple allocator approach

We can borrow a concept of Proto-Allocator from the networking TS and require type A in the std::task<T,A> to satisfy that requirement.

A type *A* meets the coroutine proto-allocator requirements if A is *Cpp17DefaultConstructible*, *Cpp17Destructible*, *Cpp17CopyConstructible*, and allocator_traits<A>::rebind_alloc<U> meets the *Cpp17Allocator* requirements, where U is an object type. *[Note: For example, allocator<void> meets the proto-allocator requirements but not the allocator requirements. —end note]*

For coroutines with two or more arguments when the first argument is of *allocator_arg_t* type, the implementation would copy-construct a proto-allocator A from the second argument. Otherwise, coroutine will use the default constructed allocator to rebind and use to allocate required memory if needed.

With new definition of task having an allocator argument

```
template <class T, class A = std::coro_allocator[1]> class task;
```

the model preserves the current behavior for stateful allocators unchanged and allows boilerplate-less use with stateless allocators.

Stateful allocator (no change)

```
// Uses default allocator
// to allocate coroutine state
std::task<int> g(string s);

// Uses provided allocator a.
template <class Allocator>
std::task<int> h(
  allocator_arg, Allocator a);

// Uses provided allocator a.
template <class Allocator>
std::task<int> f(
  allocator_arg, Allocator a,
  int x, int y)
{
  ...
  co_await h({}, a);
  ...
}
```

Stateless Allocator (boilerplate no longer needed)

```
namespace my {
  template <class T>
  using task<T> = std::task<T, MyAlloc>;
}

// Uses MyAlloc to allocate coroutine state.
my::task<int> g(string s);


my::task<int> f(int x, int y){
  ...
  co_await g("hello");
  ...
}
```

[1] Here `std::coro_allocator` is a proto-allocator that implements the behavior from P1056R1, specifically it can be copy constructed from an arbitrary allocator doing allocator type erasure internally.

3 Extend simple approach with parameter preview

The simple approach described in the previous section addresses the boilerplate concern for stateless allocators, but, it is less flexible than what is possible today for hand-crafted coroutine types, where a coroutine type designer can overload operator new of the coroutine promise and observe all arguments passed to a coroutine and decide based on those arguments how to obtain memory for the coroutine state. Here are a few examples of how this feature could be used:

`MyTask<T> MyClass::Frob();`	Parameter preview for the operator new of MyTask can observe implicit object parameter and request memory for the coroutine from MyClass
`AnotherTask<T> Broom(memory_resource &r, …)`	Use memory_resource provided to allocate coroutine state.

To support similar flexibility for standard coroutine types (std::lazy, std::generator, etc), we can add the support for parameter preview to the coroutine proto-allocator requirements. In addition to the one listed in the previous section, we can add a parameter preview requirement.

A type *A* meets coroutine parameter preview requirements if it the following declaration is valid:

A a(coroutine_parameter_preview, p_1…p_n);

Where p_1…p_n are values of some (possibly different types), and parameter_preview is a constant of type coroutine_parameter_preview_t.

Now, for coroutines with one or more arguments, an allocator is constructed with an expression A(coroutine_parameter_preview, p_1…p_n); where p_i denotes an *i*-th function parameter. Otherwise, coroutine will use default constructed proto-allocator to rebind and use to allocate required memory if needed.

4 Remove Cpp17DefaultConstructible requirement

The approaches described before have a limitation that a stateful allocator must behave in a stateless fashion if it is default constructed. This prevents usage off-the-shelf stateful allocators that may not have a default constructor. We can remove default constructible requirement, then:

- In the simple approach described in section 2, not having a default constructor would make it impossible to create a coroutine that do not have **`allocator_tag, myAlloc`** as the first two arguments.
- In the variation with parameter preview in section 3, it would mean that coroutines with arguments not recognized by parameter preview constructor won't be possible.

5 Hybrid approach

The benefit of a simple approach described in section 2 combined with removal of *Cpp17DefaultConstructible* requirement is that any type conforming to allocator requirements can be used to customize the coroutines. Parameter preview approach adds flexibility but loses the beneficial property that "off-the-shelf" allocators can be used with coroutines.

A hybrid approach would check for presence of a constructor taking coroutine_parameter_preview and one or more arguments and, if valid, would use it. Otherwise, it would fallback to the simple approach. The hybrid approach preserves the best properties of simple and parameter preview approaches.

6 Should default coroutine allocator type erase?

In the status quo of P1056R1, std::lazy<T> type does not take an allocator and the only way to provide an arbitrary allocator to a coroutine is via **allocator_arg, allocator** pair which makes type erasing the allocator unavoidable.

With the changes proposed in previous sections, we no longer have to type-erase arbitrary allocators for the std::lazy<T> with default coroutine allocator. If customer desires a type erasing allocator, it can specialize the task providing the allocator with desired behavior. This would also mean that in order to provide your own stateful allocator, a customer would need to specialize the coroutine type with an appropriate allocator. Which is a restriction compared to P1056R1, but, is in alignment with standard library types, like std::vector.

7 At a glance

Let's pick a point in the design space: hybrid approach without allocator type erasure by default and without requirement for allocator to be default constructible and examine the user experience.

Examples

```
task<int> g(string s); // Uses std::allocator<void> to allocate coroutine state.

namespace pmr {
  template <class T> using task<T> = std::task<T, polymorphic_allocator<>>;
}

pmr::task<int> f1(); // Uses default constructed polymorphic allocator.

// Uses polymorphic allocator constructed from the memory resource.
pmr::task<int> f2(allocator_tag_t, memory_resource*);

namespace my {
  struct Arena; // Some allocation arena.
  struct CustomAllocWithParameterPreview {
    template <typename… Whatever>
    CustomAlloc(coroutine_parameter_preview_t, Arena &a, Whatever const &…);
    …
  }
  template <class T>
  using task<T> = std::task<T, CustomAllocWithParameterPreview>;
};

my::task<int> h1(); // error: CustomAlloc requires Arena& to be the first arg

my::task<int> h2(Arena&); // Ok, uses CustomAlloc constructed
                          // with provided Arena argument.
```

To summarize:

- Efficient stateless allocator without boilerplate
- Comparable in flexibility to custom coroutine types with overloaded operator new
- Using type-erasing allocators is possible
- Using off-the-shelf allocator is possible.

We would like LEWG guidance on which direction to take in designing allocator model for standard coroutine types.

8 Bibliography

[P1056R1] Lewis Baker, Gor Nishanov. “Add lazy coroutine (coroutine task) type” (WG21 paper, 2018-10-07).