P3002R0: Guidelines for Allocators in New Library Classes

Pablo Halpern <phalpern@halpernwightsoftware.com>

2023-10-15 11:14 EDT

Target audience: LEWG

1 Abstract

To ensure consistency and cut down on excessive debate, LEWG is in the process of creating a standing document of design policies to be followed by default when proposing new facilities for the Standard Library; exceptions to these policies would require paper author to provide rationale for each exception. Because the Standard Library should give users control over the source of memory used by Library objects, policies on when and how classes should accept an allocator are badly needed so that allocator support does not become an afterthought. This paper proposes two policies related to allocator use, each with guiding principles and each citing examples in the current Standard. The policies proposed herein can be developed and adopted separately or as a group.

2 Policy 1: When should a class use an allocator?

- A class that allocates memory should accept an allocator at construction and retain a copy of that allocator for allocation and deallocation.
- A class that contains subobjects (base classes or members) that use allocators should accept an allocator at construction to forward to those subobjects.

2.1 Principles

1. **The Standard Library should be general and flexible**. To the extent possible, the user of a library class should have control over how memory is allocated.
2. **The Standard Library should be consistent**. The use of allocators should be consistent with the existing allocator-aware classes and class templates, especially those in the containers library.
3. **The Standard Library should encapsulate complexity**. Fully general application of allocators is potentially complex and is best left to the experts implementing the Standard Library. Users can choose their own subset of desirable allocator behavior only if the underlying Library Classes allow them to choose their preferred approach, whether it be stateless allocators, template allocators, polymorphic allocators, or no allocators.

2.2 Examples in the current Standard

The following examples both illustrate the policy and provide models how allocator-aware interfaces and behaviors are rendered in the Standard.

Classes that allocate memory from an allocator: `vector<T, A>`, `set<K, A>`, etc.

Classes that forward an allocator to subobjects: `tuple<T...>`

3 Policy 2: When should a type have an alias in the `std::pmr` namespace?

- A class template `T` having an allocator parameter `A` defaulted to a specialization of `std::allocator` should have an alias, `pmr::T` where `A` is a specialization of `pmr::polymorphic_allocator`.
- An alias `S` for such a class template that sets `A` to a specialization of `std::allocator` should have an alias `pmr::S` where `A` is set to a specialization of `pmr::polymorphic_allocator`.

3.1 Principles

1 **The Standard Library should provide simplifications of complex facilities**. The PMR part of the standard library is intended to provide a simplified allocator model for a common use case – that of providing an allocator to a class object and its subparts without infecting the object's type. Staying within the `pmr` namespace simplifies user code, as it is much easier to write and understand `std::pmr::vector<std::pmr::new_type<T>>` than `std::pmr::vector<std::new_type<T, std::pmr::polymorphic_allocator<T>>>`.

2 **The Standard Library should be consistent**. Existing allocator-aware types follow this policy, so a user would expect new ones to follow it, too.

3.2 Examples in the current Standard

Classes with default allocators: `vector`, `unordered_set`, `basic_string`, and all of the other allocator-aware containers have a `pmr` alias.

Aliases that use the default allocator: `string`, `string32`, and the other aliases for `basic_string` have `pmr` aliases.