

Document number: P0043
Date: 2015-09-27
To: SC22/WG21 LEWG
Reply to: David Krauss
(david_work at me dot com)
References: N4480

Function wrappers with allocators and `noexcept`

The Fundamentals TS v.1 introduces a new architecture for `function`, where it enforces an allocation scheme on its target objects. Because move assignment reallocates the new value, it fails to provide a `noexcept` guarantee. It also may subtly introduce bugs and breakage. The status quo is more intuitive, in terms of reference and value semantics. The new scheme also requires parallel type erasure of the allocator, costing memory and performance. This proposal formalizes the current `std::function` model with only minor extensions, and introduces a new, interoperable but allocator-aware class `function_container` to enforce allocation guarantees. `fundamentals_v1::function` fits neatly into its container model. A complete, high-quality reference implementation is provided.

1. Background	2
1.1. Containers vs. handles vs. values	2
1.2. Cost of separate erasure	3
1.2.1. Size	3
1.2.2. Speed	3
1.2.3. Flexibility	4
1.3. Dangers of reallocation	4
2. Solution	5
2.1. <code>std::function</code>	5
2.2. <code>function_container</code>	7
2.3. <code>pmr::function_container</code>	7
2.4. <code>fundamentals_v1::function</code>	8
2.5. Combination with <code>unique_function</code>	8
3. Prototype implementation	8
4. Conclusion	9
4.1. Future work	9

1. Background

When `boost::function` first gained allocator support, it followed the example of other containers, with a second template argument.¹ Later, the model was changed to follow `shared_ptr`, using type erasure to bundle self-destruct functionality with the target object, as in a shared control block.² Subsequently, when nested allocators came about, allocator erasure was conceived anew as a separate mechanism, making `function` the aggregate of an Allocator erasure and a Callable erasure.³ This major change is subtly embodied in addition of the constructor `function::function(allocator_arg_t, Allocator const &)`. However, the `assign` function, which is incompatible with the newer model, was not removed.

Since C++11, the `function` interface has thus been self-contradictory. [Clang libc++](#) implements the new allocator constructor in a degenerate way, so it only performs default construction. Otherwise it bundles allocators with targets, like Boost and `shared_ptr`. [GCC libstdc++](#) does not support allocators at all, omitting the new constructor, `assign`, and all signatures with allocator parameters. On the other hand, `fundamentals_v1::function` treats separate allocator erasure as an accomplished fact. Its synopsis still declares `assign`, but the [prototype implementation](#) does not, and the issue is not addressed by the TS or the preceding proposals.

The allocator-erasing constructor and `assign` are implementation details. On a conceptual level, the dilemma appears as a continuum of viewpoints: `function` *is* like a container,⁴ it *is not* a container (LWG 2501 comments), or it is merely “consistent with historical use of allocators” (LWG 2370). To this author’s knowledge, there has not been any comparative study of the costs and benefits of each direction.

1.1. Containers vs. handles vs. values

The critical distinction of “container-like behavior” is keeping the allocator — that is, a reference to a memory pool — as an immutable element of state separate from the target object. This allows the `function` to persistently apply an allocation policy whenever it gets a target.

Such an allocation policy allows careful management of resources. Memory pools may be used to represent strict quotas. When an object is moved from one container to another, and the containers belong to different pools, it should be deep-copied to the new pool. Allowing one pool to depend on another muddles their distinction: The dependent pool bypasses its intended limitations, and the provider pool may see its quota inexplicably depleted. The only reliable way to prevent such incoherent allocation is to permanently identify top-level containers with pools.

¹ http://www.boost.org/doc/libs/1_59_0/doc/html/function/history.html

² N2308 *Adding allocator support to std::function for C++0x*, Dotchevski (2007)

³ N2554 *The Scoped Allocator Model (Rev 2)*, Halpern (2008)

⁴ N3916 *Polymorphic Memory Resources - r2*, Halpern (2014)

Not every container wants to implement an allocation policy, though. Some containers merely aggregate views to preexisting, perhaps shared, objects. For example, functions in a callback registry continue to belong to the registrant. Arguments bound within the function object should not be reallocated into the registry pool simply because they were bound by value. In this case, the pointer-like behavior of `std::function` is appropriate.

Other `std::function` objects are not retained anywhere in particular, but are simply treated locally as callable values. When allocation is a forgotten minor detail, reallocation is potentially most surprising. Since function objects are conventionally passed by value, and local function reassignment carries no gotchas, reallocation is a potential pitfall.

In summary, three use cases can be distinguished by the relationship of the wrapper to the target:

- Some `functions` act as persistent containers with a well-defined resource management policy.
- Some work like smart pointers to preallocated resources.
- Some are naively used as local variables, as if they were merely views to a pure value.

Container-like behavior benefits the first case but complicates or harms the latter two.

1.2. Cost of separate erasure

Current `function` implementations enable allocator erasure to the extent of allowing an object to clone or destroy itself, using the allocator that created it. This is as simple as having `clone` and `destroy` functions alongside the more familiar `invoke`. A second erasure, in the form of a `memory_resource *`, is needed if the allocator may exist in the absence of any target (or must persist in the presence of a target stored within the wrapper). This carries costs in terms of memory, speed, and support for allocator functionality.

1.2.1. Size

The size of current `function` objects is mostly determined by the standard's recommendation that the small target optimization should support a `bind` object containing a PTMF and an object pointer ([`func.wrap.func.con`] N4527 §20.9.12.2.1/10). This typically takes three pointers; add a pointer to the erasure structure (e.g., `vtable`), for a total of four machine words. Such is the memory layout of `boost::function`.

Adding a pointer to an external `memory_resource` would ordinarily take one more word, but `fundamentals_v1::function` additionally supports classic allocators by transparently applying `resource_adapter`. The reference implementation takes ownership of the resulting object via `shared_ptr`. This adds the overhead of a control block allocation even when the argument is an external `memory_resource` in the first place. The wrapper itself may also grow by up to two words (50%) to accommodate the `shared_ptr`.

1.2.2. Speed

`experimental::memory_resource` is polymorphic, so it incurs indirect function calls. Whereas binding a new target object currently works without an indirect call, separate erasure needs one to dispatch an `allocate` call. The same applies to move-assignment. Whereas these

operations could likely have been inlined, the allocator might instead be hidden behind a mispredicted branch.

Knowing its own allocator, a target can clone itself using one indirect dispatch. Separate erasure adds another. When an object is copied or moved to a new pool, it is usually still to an allocator of the same type, so the operation can be done with no further indirect calls. Separate allocation adds one dispatch per allocation and deallocation. The prototype implementation uses an additional dispatch to get the memory resource pointer, before any indirect calls to it.

1.2.3. Flexibility

Allocators are inherently template-oriented, and templates are resistant to type erasure.

`fundamentals_v1::function` flattens classic allocators into type-erased allocators (N4480 §8.3, §8.7), losing much functionality. Unlike the similar facilities in namespace `pmr`, which require the user to apply `resource_adaptor` for manual flattening, this is applied implicitly and potentially surprisingly.

- Instead of using `rebind` to ensure proper alignment and allocation size, `resource_adaptor` rounds up the size and strips padding as by `std::align`.
- Any behavior specific to a particular type is lost because the allocator is rebound to `char`.
- The allocator object is used to make a copy of itself in a `shared_ptr`, instead of simply being embedded in the wrapper or in the same erasure as the target. This may confound allocators that only expect a certain number of allocations, or a particular order to deallocations, or a particular size range. (This applies even to stateless allocators. Note that it is specific to `fundamentals_v1::function`, and not part of `resource_adaptor`'s behavior.)
- Fancy pointers are replaced by native pointers.
- Instead of using `construct` and `destroy`, it applies default behavior.
- The `propagate_on_container_*` and `select_on_container_copy_construction` traits are replaced by the defaults.

Custom allocators may be sensitive to the exact types of objects being allocated and their lifetimes. The `resource_adaptor` compatibility layer is more of a stopgap to allow general-purpose allocators to be migrated to the `memory_resource` model, and less a pillar of support for high-performance, application-specific allocators.

1.3. Dangers of reallocation

In addition to the overheads surveyed in the preceding section, uniform use of container allocation semantics may lead to errors and vulnerabilities. Contrary to the intent of better-organizing allocation, functions written without container semantics in mind, but only value- or smart-pointer semantics, may end up scrambling resources among pools and storing resources in the global free store (which is free of quotas) instead of the intended pool.

For example, `map<string, fundamentals_v1::function<void()>>::operator[]`, a typical dispatch table structure, will default-construct functions when adding entries. A default-constructed function carries a sticky association with the global free store. So, using this method

to add an entry will remove its target from any memory pool, whereas `std::function` and `boost::function` preserve allocation. Instead, `insert` must be called, and `erase` before any subsequent reassignments.

Swapping `fundamentals_v1::functions` (or other containers) with different allocators has undefined behavior. It may lead to an assertion or a resource paired with an allocator that cannot release it. Unlike the previous example, allocators are preserved when inserting functions into a `priority_queue<pair<int, fundamentals_v1::function<void()>>>`,⁵ such as may be found in a task scheduler. If two tasks come from clients with different memory pools, the first swap operation to touch them will terminate the program. To fix this, the structure must be rewritten to store the functions in a node-based container, which may add overhead.

Such problems may also occur in ordinary functions or classes besides sequence containers, which happen to assign to `function` as a value type without considering allocation.

2. Solution

The proposed solution has two parts: a new `function_container` class template, and improvements to `std::function` that allow it to be interoperable, i.e. to share type-erased target objects between corresponding `function` and `function_container` specializations.

This write-up is based on the [cxx_function](#) library,⁶ which is intended as a reference implementation and released freely under the MIT license. The library also implements other features which are covered in other proposals. Nevertheless, it is more efficient than its `libstdc++` or `libc++` counterparts.

2.1. `std::function`

This proposal refines the allocator semantics of `function` within the intersection of the standards and the implementation practice, which is to say:

- An implementation such as `libstdc++` or `libc++` can adopt this specification without breaking their users' code.
- The changes may be applied retroactively to C++11 and C++14 modes while still conforming to those standards.

The status quo since N2308 is based on the allocation scheme provided by `shared_pointer` via `allocate_shared`. From this basis, shared ownership was removed, and cloning was added. In terms of ownership and allocation semantics, `function` essentially fits into the category of smart pointers.

Functionality is extended beyond the status quo in the direction of comprehensive support for allocation functionality via `allocator_traits`. The changes are summarized below.

1. Restore the `noexcept` guarantee to the move assignment operator.

⁵ Assume that a comparator is provided to sort by the `int` values. Such a structure can be adjusted to prevent overflow issues.

⁶ https://github.com/potswa/cxx_function. Scroll to `#allocators` for more information.

2. Deprecate the constructor `function(allocator_arg_t, const A& a)`, and require it to be equivalent to the default constructor.
3. Specify that member functions without an `allocator_arg_t` parameter, which take a new target object, behave as if `allocator_arg_t` was passed with a default allocator.
4. Require the target object lifetime to be managed by the `allocator_traits::construct` and `destroy` functions. Do not require a copy of the allocator object to be kept if `destroy` behaves equivalently to the default `allocator_traits` implementation.
5. Remove the `std::uses_allocator<std::function<Sig>, A>` partial specialization to consistently avoid reallocation. (This is a nominally a breaking change, but neither `libc++` nor `libstdc++`, at least, implement the feature: `libc++` does not specialize `uses_allocator` and `libstdc++` lacks the constructors to make it meaningful.)

For the following rules, treat any `function_container` parameter as if it were an ordinary function. When used as the source operand in construction or assignment, the persistent allocator of a `function_container` is ignored and only the target object is considered.

6. Specify that the copy constructor and copy assignment operator use a copy of the allocator which originally created the target object.
7. For adopting a target object from one wrapper to another, specify that when an `Allocator` parameter appears with a target object parameter of the wrapper type, and the parameter's target object is managed by an allocator:
 - If the type of the new allocator and the type of the allocator used to create the target, both rebound by `rebind_alloc<char>`, are the same type, and those two allocators compare equal, and the target object parameter was not passed by lvalue reference, then the target object shall be transferred to the destination wrapper and no exception shall be thrown.
 - Otherwise, if the first condition is met (same allocator type after `rebind`), a new target object is created. The allocator used is the value of the `Allocator` parameter, converted first to `rebind_alloc<char>`, and converted again to the type of the allocator used to create the original target. The object is copy- or move-initialized from the original target. If an exception is thrown, the source object is left unchanged, and (for an assignment operation) the destination wrapper receives a value of `nullptr`. This implies that a target's throwing move constructor will be ignored in favor of its copy constructor.
 - Otherwise, if the allocator parameter is incompatible with the allocator used to create the original target, an exception of type `allocator_mismatch_error` is thrown.

If N4543 *A polymorphic wrapper for all Callable objects* is accepted, its in-place construction functions receive an additional guarantee:

8. If the in-place construction type tag specifies the same type as the wrapper, then a temporary is constructed according to the in-place argument list, and the wrapper is move-constructed from that temporary.
However, if there are several allocators of the same type in the same parameter list, the implementation is free to skip move operations and construct the target into the final destination pool.

2.2. `function_container`

A persistent allocator may be added to a `function` by aggregating them both together, as the above refinements are sufficient to implement the allocator-aware container interface ([`container.requirements.general`] N4527 §23.2.1/16). The `function_container` implementation simply wraps `std::function` while ensuring that the encapsulated allocator is passed to every `std::function` member that can accept it, and applying the allocator propagation traits as usual. Member functions that conflict with this goal, namely `assign`, and given N4543, `allocate_assign`, are not exposed in its interface. The constructor accepting no target, but only an allocator, works and it is not deprecated. However, allocator parameters are of the given allocator type. Any allocator type erasure is up to the allocator itself.

The move assignment operator maps to `assign` on the encapsulated wrapper and may cause reallocation. This behavior, and its `noexcept` guarantee, depend on `is_always_equal` and `propagate_on_container_move_assignment`, as in other allocator-aware containers. Assignment from `std::function` may throw `allocator_mismatch_error` and never respects POCMA or POCCA. Move-assignment *to* `std::function` is `noexcept`, as specified in the preceding section.

Being an allocator-aware container, it implements `allocator_type` and `get_allocator` members. Thus it supports uses-allocator construction and `scoped_allocator_adaptor`. The embedded allocator is rebound by `rebind_alloc<char>`, similarly to a node-based container rebinding to the node type, except that rebinding occurs again upon each type erasure operation.

When the erasure contains an allocator object, it coexists with the persistent one in the wrapper. Allocation and construction are specified (see above) to be done by the allocator object within the target object erasure. After each operation, the state of the the erasure's allocator is copied back to the container, to give the impression that the container's allocator object is doing the work. In any case, the two allocators always compare equal, and their values never change according to the equality operator.

For compatibility with overloaded function, with a list of signatures, the allocator is added at the beginning of the template parameter list.

```
template< typename alloc, typename signature >           // This proposal alone.
class function_container;
```

```
template< typename alloc, typename ... signatures >     // With P0045.
class function_container;
```

2.3. `pmr::function_container`

A polymorphic memory resource may be combined with `function_container` using `polymorphic_allocator`. This provides uniformity between `function` and the other polymorphic allocation facilities.

```

namespace pmr {
template< typename sig >
using function_container = function_container
                        < polymorphic_allocator< char >, sig >;
}

```

The name `pmr::function_container` is chosen over `pmr::function`, to reflect its behavior. Like the other `pmr` aliases, this binds and removes the allocator template parameter.

2.4. `fundamentals_v1::function`

If desired, `function_container` may also be used as the basis of a conforming, and optimal, implementation of `fundamentals_v1::function`. The key is a new allocator class enclosing a `shared_ptr<memory_resource>` instead of a raw pointer. It could not purely be a `function_container` specialization due to the `get_memory_resource` member.

2.5. Combination with `unique_function`

N4543 orthogonally proposes a polymorphic call wrapper `unique_function`, not requiring target copyability or movability. Applying allocator semantics to `unique_function` produces `unique_function_container` and `pmr::unique_function`. Non-copyable targets cannot be transferred to copyable wrappers, so this proliferation is unavoidable.

Reallocation requires a copy constructor or a `noexcept` move constructor (see above, §2.1/6.2) but it never occurs if `is_always_equal` is `true_type`. Targets are checked for suitability against these constraints, and any constructor, assignment operator, or `assign` overload is removed by SFINAE when a target is unsuitable.

`unique_function_container` is actually likely to be more commonly appropriate than `function_container`, because most carefully-allocated targets are not supposed to be copied.

3. Prototype implementation

The [`cxx_function`](#) implementation adds allocation flexibility with no compromises. It performs on par with `libstdc++` and `libc++` in the public *CxxFunctionBenchmark*,⁷ but it uses wrapper storage space more efficiently and applies optimizations more generally in the presence of allocators (which are not benchmarked).

Each erased type is described by a global tuple of values (target `typeid`, allocator `typeid`) and function pointers (destroy, destructive-move, clone, call). When a function is trivial, its pointer is set to `nullptr`, allowing the wrapper to avoid an indirect call. The default allocator (and others with `is_always_equal` and no fancy pointers) allow trivial destructive-move. In most cases, the `function` move constructor simply calls `memcpy`.

⁷ <https://github.com/jamboree/CxxFunctionBenchmark/>, no relation to `cxx_function`.

Its internal organization isn't perfect: `function_container` should be implemented by wrapping a `function` subobject, but the library derives them both from a common base class. This is an artifact of its evolution, and it is fixable.

4. Conclusion

The allocation policy of `std::function` is an obscure topic, but it does matter in practice. It has come up on StackOverflow ⁸ and (at least indirectly) on `std-proposals`: There are users who have encountered the limits of current implementations. It is an important consideration for programs that are miserly with memory or CPU cycles, and intensive on closure semantics.

Without `function_container`, the standard library still contains useful tools to face the challenge of taming target object allocation, particularly given the proposed refinements to `std::function`. However, complete and optimal allocator support involves countless nuances, and interoperability with ordinary, value-semantic `std::function` is impossible without standard library support. It is essential to standardize `function_container`, at least in a TS.

`std::function` is a “vocabulary type;” `function_container` will never be. They are analogous in obscurity to `std::string` and `std::basic_string`: most users will remain blissfully unaware the latter. Interoperability is provided though, so unlike allocator-aware STL containers, encountering `function_container` is not an unpleasant surprise. A typical user only needs to know that it assigns to and from `std::function`. The potential for harm is low.

4.1. Future work

The same principles can be applied to other value-handle classes, such as `std::any` and the likes of `std::unique_ptr`. Users should be able to allocate an object in one pool and then assign it to another pool without manually comparing the two pools.

The dichotomy of container-like and value-like handles is the only way to have an intuitive, reference-semantic move assignment operator simultaneously with the allocation safety of `scoped_allocator_adaptor`. This is a promising future direction.

It may be possible to factor `function_container` into a class template such as `erasure_container_adaptor`, capable of working with `any` or user-defined classes. Only an attempt at implementation would tell.

⁸ [catch std::function allocations at compile time](#), [How can I create a std::function with a custom allocator?](#), [Does std::function support a custom allocator?](#), [... constructor with custom allocator but no other args?](#), etc.