# Support for static and SBO vectors by allocators

Document: P2667R0

Date: 2022-10-09

Project: Programming language C++

Audience: LEWG(I)

Reply-to: Bengt Gustafsson, bengt.gustafsson@beamways.com



## Introduction

This proposal mandates that `std::vector` must actually use `allocate_at_least` for its allocations and abide by some simple rules for its allocation pattern. `std::vector` must also handle moving of individual elements in more situations in move constructor and move assignment.

The proposal also contains allocator traits that allows allocator clients to optimize storage when a static allocator is provided.

Also proposed is a new container constructor and assignment operator between containers of the same T but different allocator type. This is essentially a separate feature but with more allocators to choose from it will be more needed.

A `buffered_allocator` template which has an internal buffer and a backing allocator for overflow is also included in the proposal, as well as some trivial non-allocating allocators useful for implementing `static_vector` with different overflow policies.

# Motivation and scope

Introducing static_vector and small buffer optimization-enabled vector (sbo_vector) has been a long standing request in C++. With P0401 now in C++23 it is *almost* possible to implement such vectors using only custom allocators with internal buffer storage. What is missing is that there is no standard requirement on `std::vector` to actually use the `std::allocate_at_least` function, and there are no guarantees when it comes to the allocation pattern that vector is allowed to use its allocator for. However, if `std::vector` continues doing what the current implementations do, and uses allocate_at_least for all allocations, static_vectors and sbo_vectors can be implemented using allocators, with just one further requirements on vector: The rules for when elementwise moving is required when vectors are moved must be updated.

However, even with this vector with a fixed max_size buffer still wastes memory as it will store the buffer address, size and capacity information, while only size is needed, and most often the size can fit in an 8- or 16-bit integer. To remedy this a new allocator_trait is needed. (To avoid problems if the entire allocator_traits has been partially specialized for some allocator type we propose this as a constexpr variable template which uses the same strategy as for `allocate_at_least`). With this a standard library implementation can optimize the storage requirements for such `std::vector` specializations.

The advantage of this strategy over creating new `static_vector` and `sbo_vector` containers from scratch is to a large extent that generic code that expects a `std::vector` specialization can use the new containers without having to be further generalized. Another advantage is that it increases composability between containers and allocators. It also reduces the length of the standard text to not have to repeat all members of vector two more times, although this is of course offset by the text required to specify the mechanisms proposed here.

# Impact on the standard

This proposal has smallish impact on the standard library in a few different places. The rules that are marked specifically as applying to std::vector would apply similarly to other containers with a single growing block strategy.

## std::vector must allocate using allocate_at_least

And in addition it must set its capacity to the returned capacity in the allocation_result object.

This requirement emanates from the fact that a sbo- or static- allocator only has *one* buffer of the set size. Thus they can return this buffer and its size from the first allocate_at_least call, and the vector can use this as long as it is large enough. If the vector would call allocate() according to its normal size increase policy another block of small enough size could be requested and the allocator would return its internal buffer again, as it does not keep track of whether it is in use. The vector would have to special treat this and not move the elements if both buffer addresses are equal. This seems to be more intrusive than requiring that allocate_at_least is called.

## std::vector must only allocate larger blocks than it already has

This is consistent with how vector's growth strategies work today, and anything else is infeasible as vector's iterators are contiguous. With this rule put on paper a buffered allocator can do allocations outside its buffer if the parameter to `allocate_at_least` is larger than its own buffer's capacity (as the first allocation returned the buffer capacity and vector's don't allocate new blocks until previous capacity is exhausted).

The only `std::vector` method which is allowed to reduce the size of the allocated block is `shrink_to_fit`. A library that implements this needs a fix to avoid shrinking more if the data is already in the internal buffer. This can be done by checking if the capacity returned from allocate_at_least for the vector size was the same as the previous pointer. Note: This situation can't be ignored as vector would otherwise in place construct the elements onto themselves.

## std::vector must move elements if in source's internal buffer

The vector must amend its rules for when to move elements so that whenever the source's size is lower than the source's allocator type's value for `allocator_info::buffer_capacity` the elements must be moved rather than the pointers.

```
bool move_elements = src.capacity() <=
std::allocator_info::buffer_capacity<SrcAlloc>;
```

## New traits to enable static_vector size storage optimization

A pair of variable templates are introduced to handle the differences needed between regular, SBO and fixed categories of allocators. The author proposes that these are placed in a *namespace* fulfilling the same function as allocator_traits. This prevents from specializing all *current* traits at once by specializing the class template itself. The allocate_at_least function should preferably be moved into this new namespace before the release of the C++23 standard.

```
namespace std::allocator_info {
    template<typename Alloc> constexpr size_t buffer_capacity = 0;
    template<typename Alloc> constexpr bool can_allocate = true;
}
```

As long as the can_allocate value is true for the allocator that a std::vector is instantiated with, no optimizations are possible. The allocator will sometimes at least allocate using operator new or other means which return arbitrary addresses, which must be stored by the vector object along with information about the current size and capacity.

However, if the `can_allocate` value is false the allocator offers the vector certain guarantees:

1. `allocate_at_least` for any element count <= buffer_capacity returns the internal buffer pointer and the buffer_capacity. If element_count > buffer_capacity allocate_at_least does whatever is appropriate at overflow.
2. `allocate` for *any* element count returns the internal buffer pointer.
3. `deallocate` does nothing.

This works for an unspecialized vector or other container as long as it follows the prescribed rules, i.e. never calls allocate().

These guarantees however *allows* vector to optimize its storage to only a size member, apart from the allocator object. The number of bits required for this size member can be inferred from the buffer_capacity of the allocator.

- The data() method of the specialized vector can call allocate(0) to avoid having to store the address of the current buffer. Thanks to #2 this does not involve any if statements being executed by the allocator.
- The capacity() method can return the `allocator_info::buffer_capacity` value directly to avoid having to store the count returned from allocate_at_least.
- size() returns the stored size member.

If other functions of vector are implemented in terms of these basic functions no further specialization is needed, except for an internal function to set the size.

## Overlaying sbo_vector capacity with buffer

A sbo_buffer that minimizes its storage requirements could be implemented by using a *overflow* value in the size member of the optimized static_vector storage model. If the size is set to this value the buffer is repurposed to the three pointers needed for the overflowed case. The drawback with this is that all methods accessing the vector contents must first check for this special value to know how to find the buffer address. This can be slow, but if all access is using iterators not so bad.

Implementers can explore such options, this proposal only makes them possible.

## Iterator invalidation guarantees for vector are affected

The iterator invalidation guarantees when moving vector contents from one vector to another will have to be changed to not guarantee stability for all allocators.

The rule is that if `allocator_info::buffer_capacity > 0` the iterator guarantees are changed in addition to the propagate_on_container_move/copy_assignment special case already stated in the iterator invalidation guareantees.

## Copying and moving between containers with different allocators

std::vector currently does not allow copying and moving between containers with the same element type but different allocator types. This seems like a useful feature to have, although the new constructors probably needs to be explicit to avoid surprises.

This feature becomes more important considering copying and moving between vectors with different size buffers, but with the same backing allocator, and vectors with the backing allocator itself. In these cases it is beneficial to move the data pointers instead of the elements, provided the source's internal buffer is not in use, as described above.

Detecting if the backing allocator type is the same is uses a new trait `backing_allocator_of<Alloc>` which by default is Alloc but for buffered_allocator is its third template parameter:

```
namespace std::allocator_info {
    template<typename Alloc> struct backing_allocator_of {
        using type = Alloc;
    };

    // Specialize for a buffered allocator described below.
    template<typename T, size_t SZ, typename Alloc>
    struct backing_allocator_of<buffered_allocator<T, SZ, Alloc>> {
        using type = backing_allocator_of<Alloc>;       // Recurse if necessary
    };

    template<typename Alloc> using backing_alloc_of_t =
backing_alloc_of<Alloc>::type;
}
```

Then the move constructor and move assignment can check if the backing_allocator_t of of the lhs and rhs are the same, and if so move the buffer ownership to the destination.

Standardizing this trait allows for user defined allocators with similar semantics as buffered_allocator. If only `buffered_allocator` is ever going to be able to optimize moving/copying with the same backing allocator this trait can be internal to vector. he author thinks that even as we today have a hard time coming up with more than one way of buffering elements that are to be accessed by a contiguous iterator, we should not limit the future by not standardizing this trait.

## Preventing use for other containers

Allocators with `std::allocator_info::buffer_capacity<Alloc> > 0` are not useful for containers which have other allocation patterns than vector, basic_string and similar types. Thus it would be preferred to have static_asserts in such containers to get a good error message if such an allocator is used for instance in a map.

## Rebinding rules

In node based containers and for the Microsoft debug mode iterator integrity check blocks the allocator provided as template parameter is rebound to another T, such as `map_node<T>` for some implementation of std::map. It is unclear at this point exactly what the uses for rebinding allocators is.

This proposal does not give thought to this but posits that the rebinding rules work as usual, so that for a static_vector<T, 100> the Microsoft control block allocator would allow for 100 control blocks inside the vector's memory footprint. Microsoft would have to get the backing allocator and rebind that to get similar behaviour as today.

## ABI considerations

As this proposal does not change behaviour or data layout of vectors which use pre-existing allocators there are no ABI problems for currently existing code.

However, an implementation must implement the optimizations outlined above for the `can_allocate == false` case *immediately* to avoid future ABI breakage when those optimizations do get implemented.

The only further caveat is that any implementation changes made to optimize the implementation for `can_allocate == false` must not change the non-static data member layout in vectors using other allocators.

## Compile time considerations

Adding new allocator classes and traits to `<memory>` and adjustments in `<vector>` will add some compile time even if the new allocators are not used. As Modules get more and more used this impact will be reduced greatly. If the alternative is static_vector and sbo_vector as separate class templates the added code would be much more, but on the other hand maybe those were intended for their own separate header files which are not included by as many translation modules.

## Included allocator classes

This proposal contains a set of allocator classes which model the same *Allocator* concept as std::allocator (in its C++23 form). The main class is buffered_allocator which adds a buffer of a specific number of elements to a backing allocator and forwards most of the properties of the backing allocator. This includes can_allocate, importantly.

```
template<typename T, size_t SZ, typename Backing = allocator<T>> class
buffered_allocator;
```

In addition three trivial non-allocating classes are proposed, which allow customizing the overflow behaviour of a static_vector.

```
template<typename T> struct terminating_allocator;      // Terminate in alloc()
template<typename T> struct throwing_allocator;         // Throw bad_alloc in
alloc()
template<typename T> struct unchecked_allocator;        // Return nullptr from
alloc.
```

The `unchecked_allocator` case still not offers the preferred solution as the if statement in `std::vector::reserve` still has to exist if vector does not know that it deals with an unchecked allocator. One can hope that an optimizing compiler will be able to optimize this away but it is unclear if this is realistic. If this is not true a further trait for ignoring overflow may have to be added to single out the ignoring allocators. As there are hardly more ways to ignore overflow than using unchecked_allocator offers it would also be possible to specialize vector on the fact that the backing allocator is a specialization of unchecked_allocator, ignoring custom allocators that also may have this behaviour.

## Included type alias templates

To simplify usage of vectors using buffered_allocator with different backing allocators a set of type alias templates are included in this proposal. These type aliases should be in the `<vector>` header. The reason for providing three different static_vector type aliases is the lingering uncertainty on which overflow policy should be default. If this is resolved it could be better to only have one `static_vector` type alias and let programmers create the other two as needed.

The author believes that the unchecked variant is most appropriate to be this default policy. The rationale is that for programs with normal safety requirements (programs will not break pre-conditions) this is the appropriate selection. Programmers using static_vector will check for overflow by other means outside the static_vector and *don't expect* static_vector to do any checking for them.

```
template<typename T, size_t SZ, typename Backing = std::allocator<T>>
using sbo_vector = vector<T, buffered_allocator<T, SZ, Backing>>;

template<typename T, size_t SZ>
using static_vector_throw = sbo_vector<T, SZ, throwing_allocator<T>>;

template<typename T, size_t SZ>
using static_vector_terminate = sbo_vector<T, SZ, throwing_allocator<T>>;

template<typename T, size_t SZ>
using static_vector_unchecked = sbo_vector<T, SZ, unchecked_allocator<T>>;
```

# Technical specification

The wording changes for this seem to be rather obvious from the description above.

# Future directions (NOT PROPOSED)

## Usage for basic_string

std::basic_string has a suitable allocation strategy it is not proposed to be included it as most implementations already have their own small buffer optimization in place. An alternative would be to mandate that basic_string must forego its own small buffer optimization (if any) in case `std::allocator_info::buffer_capacity<Alloc> > 0`. This is not proposed.

### flat_map family and other container adaptors

With the flat_map family of class templates in C++23 combined with the above sbo and static vectors similar behaviour can easily be achieved with just some type alias templates such as:

```
template<typename Key, typename T, size_t SZ, typename Compare = less<Key>>
using sbo_flat_map = flat_map<Key, T, Compare, sbo_vector<Key, SZ>,
sbo_vector<T, SZ>>;
```

However, with already four containers in the flat_map family it seems a bit over the top to declare 16 type aliases like this, consuming 16 names in the std namespace. If stack, queue and priority_queue are included this is another 12 type aliases.

Therefore no such type aliases are proposed.

## Possible addition for deque

One container that doesn't fit in any of these categories is deque, as it presumably has one vector-like block and then any number of equal size blocks. What would be interesting to have is maybe a deque specialized to use a rebind of the incoming static- or sbo-allocator for the main block and then use the `backing_allocator_of<Alloc>` allocator for the individual blocks, or if this has `!can_allocate<Backing>` the static case std::allocator. The need for a deque that can

only allocate a fixed number of blocks allocated on the heap seems limited, but a deque which keeps the first block pointers in an internal buffer until it is exhausted seems like a fair deal.

## Allowing use for node based containers

With an extended set of traits and a bitmap of used elements a static- or sbo- allocator could be devised for node-allocating containers such as maps and linked lists. This is under the assumption that instead of calling allocate_at_least those containers would call allocate(1) each time a new node is needed.

The allocators useful for these containers are not optimal for vector as they have to keep track of which elements are in use, and the move rules for the sbo_case are more complicated, as the container itself may have ot move a subset of its nodes (those in the SBO buffer) and not others (those on the heap).

This breaks not only the iterator invalidation rules of today, but also the node data address stability guarantees, which makes this less attractive.

# Acknowledgements

Thanks to my employer ContextVision AB for supporting the author attending standardization meetings.