A Direction for Vector

Document number: **P3147R1**
Date: 2024-03-10
Audience: Library Evolution Working Group
Reply to: **Alan Talbot**
cpp@alantalbot.com

Abstract

This paper proposes a direction for development of a new contiguous variable-size sequence container with storage and element management flexibility. This container is meant to eventually replace `std::vector` as the "go-to" container, offering everything vectors do and more (except `vector<bool>`!). In the near term it could inform or even subsume P0843: `inplace_vector` and P0274: clump (but see Objections below).

The idea is to have one container with several performance-related behavior adjustments that allow the user to fine-tune the container in place based on profiling measurements, with no other changes to the client code. Within the bounds of physical limitations, any settings should work in any context—albeit with widely varying performance (both of memory and runtime), and perhaps with different exception guarantees.

The design spans two orthogonal dimensions:

- Location of storage: local (embedded/stack), dynamic (heap), or both (SBO).
- Management of storage: top (vector), bottom (stack), or middle (deque).

It offers a choice between fixed and dynamic storage (independent of storage location, thus allowing fixed-size dynamic allocation). It also offers control over dynamic growth with a choice of linear growth specified in elements, or exponential growth specified as a fraction.

The goal of this proposal is to measure the interest in this approach, gather information and feedback, and especially collect objections and their possible resolutions.

Revision History

R1: Added user control of the size member type. Changed the recommendations for P0274. Made a couple of other small improvements to the text (but not to the meaning).

Motivation

Vector is probably the most-used container in the Standard Library, and it is pretty much always chosen as an example of a vocabulary type and a prototypical example of what *should* be in the Library. Like all the containers, it does its exact job very well, but if you want something slightly different, you have no recourse within the Library. Here is a short list of some of the things not available in or not well supported by the Standard Library:

1. A local vector (e.g. `inplace_vector`).
2. A small buffer optimized vector (e.g. clump/small vector).

3. A way to control vector growth rate.
4. `push_front` for vectors. (I proposed this in P0563, but it was not well received because of a perceived philosophical conflict with the meaning of a vector.)
5. A stack that grows up.
6. A contiguous and/or local deque.

Attempting to provide all of this missing functionality with multiple distinct containers seems impractical at best. It would generate too much work for the Committee, too much ink in the Standard, and the resulting set of options would be too confusing for users. (I have found it is difficult enough to get programmers to understand and effectively use the tools we have today.)

The interactions between such containers would be an even bigger challenge. It seems like efficient operations (moves, comparisons, etc.) between different location and management strategies are an important feature. I'm sure it's possible to do this with distinct containers, but it is surely quite a lot more difficult and confusing.

Client code compatibility seems like a very important goal—we want users to be able to switch between various storage and element management strategies without worrying about breaking their code. The difficulty of meeting this requirement will expand exponentially with each added container. Both P0843 and P0274 refer to `std::vector` and express or imply a desire to match it "where possible". Providing a single new container eliminates most of that problem, especially for new code and for situations where `std::vector` can be replaced by the new sequence. On that topic, I believe it *will* be possible to match vector behavior with this sequence so that a direct substitution can be done in most cases.

Existing Containers

Here is a quick review of the sequence containers we have today and their specific limitations.

Arrays

Arrays, both built-in and `std::array`, are a type of contiguous sequence, and they have storage location flexibility. Because they are fixed-size, arrays end up being useful for a different set of problems than variable-size containers. (I have often used an array to model a variable-size embedded container by using a sentinel value—but that's awkward and error-prone at best.)

vector

`std::vector` is a dynamically allocated, variable capacity contiguous sequence, but it has no controls other than supporting custom allocators, and its storage cannot be local. It's worth mentioning that a custom allocator is not a practical way to control memory usage for most users.

deque

`std::deque` is a dynamically allocated, variable capacity sequence that offers efficient front and back operations, but it is not contiguous and is extremely inefficient for small sizes, and it cannot be local.

Node-based Containers

The node-based sequence containers (`std::list`, `std::forward_list`) are again designed to address a different set of problems than vector. They are very inefficient for small objects, and of course the storage is not contiguous or local.

Objections

Derailing P0843 inplace_vector

I am very excited by the `inplace_vector` proposal. It addresses one of the biggest limitations of vector, and it is well along in the approval process. An obvious objection to this proposal will be: are we going to drop or delay P0843? I think that unless there is very strong support to do so, the answer should be no.

I believe that almost all of the considerable work that has gone into P0843 will be directly applicable to a new sequence container. One way to think about this proposal is as an expansion of the scope of P0843 (and P0274). Little if any of this work would be discarded regardless of how we proceed.

If this proposal is deemed too ambitious for C++26 (which seems likely), then I believe we have a way forward that does not create future maintenance issues beyond the unavoidable one with `std::vector`. The idea is to work on a design such as outlined in this proposal and allow it to inform the design of `inplace_vector`. The goal would be to eventually replace `inplace_vector` with a using declaration once we have a new more flexible sequence. This approach does not reduce the additional burden on P0843 to zero, but it is far less work than expanding it to a fully flexible design.

Derailing P0274 Clump

I am also excited by the clump (small vector) proposal. It addresses another big limitation of vector, but so far it has not progressed in the approval process. If P0274 does not move forward for C++26, then I recommend subsuming it into a new sequence proposal. If by any chance it does move forward for C++26, the answers are the same as for P0843.

Implementation

Another valid objection is that a concrete proposal with no implementation experience may be too new to put into C++26. In contrast, local vectors and small vectors have years of prior art in Boost and elsewhere. I agree with this objection. I think that unless there is such a strong interest that a lot of people are willing to get involved, the solution is to progress P0843 as I have described above, and work on a more flexible sequence for C++29.

Overgeneralization

I have heard objections to this type of design, basically that each problem should be solved with a dedicated tool. I think that there is an easy answer in this case: we don't have the time and resources to add half a dozen new containers to the Standard Library, nor do we want to take on the maintenance load that would entail. I also think that this really is one (albeit broad) problem. The proposed design offers a single solution to that problem which can be adjusted in various ways to achieve optimum performance.

Standardizing a number of distinct containers would require users who want to switch between them to pick names for a bunch of type aliases (one for each use). It would fall on us to ensure that all these types are compatible without client code changes. (Achieving compatibility would likely require changes to **`std::vector`**.)

Design Overview

Basics

The sequence object itself is always local (that is, on the stack or embedded in another object by composition). Elements are always stored contiguously. There is a size and a capacity, either of which may be zero. The size cannot be greater than the capacity.

Template Parameters

The first template argument provides the element type.

The second template argument is a non-type structural argument that provides a set of traits specifying the user-defined behaviors of the sequence. The default has been chosen to provide equivalent behavior to **`std::vector`**. It might look something like this:

```
template<std::unsigned_integral SIZE = size_t>
struct sequence_traits_t {
   using size_type = SIZE;    // The type of the size field.
   bool dynamic = true;       // True if storage could be dynamically allocated.
   bool variable = true;      // True if the capacity can grow.
   size_type capacity = 0;    // The size of the fixed capacity (or the SBO).
   enum {FRONT, MIDDLE, BACK}          // I.e. vector, deque, stack.
   location = FRONT;
   enum {LINEAR, EXPONENTIAL, VECTOR}  // See below.
   growth = EXPONENTIAL;
   size_t increment = 0;  // The linear growth in elements (size_t, not size_type)
   float factor = 1.5;    // The exponential growth factor (> 1.0).
};
```

For an allocator-aware version, the third template argument would be the allocator type (see Allocators below).

Memory Allocation

The **`size_type`** type provides control over the type used to represent the size of the sequence. This allows better utilization of the fixed capacity in the common case where its size is small and the elements are small. For example, on a typical 64 bit machine, a sequence object size of 24 bytes can hold 16 chars (in the local SBO buffer) if the **`size_type`** is **`size_t`** (the default) but 23 chars if the **`size_type`** is **`unsigned char`**. This parameter is constrained to an unsigned integral type.

The **`dynamic`** switch specifies whether the capacity is local or dynamically allocated. A local sequence contains its capacity in the sequence object. The capacity of a dynamically allocated sequence may reside in the sequence object or in a dynamically allocated memory block (depending on whether a local SBO buffer is in use).

The `variable` switch specifies whether the capacity is fixed or variable. A variable capacity sequence can grow indefinitely, bounded only by physical limitations. The `capacity` value specifies the size of the fixed capacity in elements.

A local sequence must have a fixed capacity. A dynamically allocated sequence may have either a fixed or variable capacity. If a dynamically allocated, variable capacity sequence has a non-zero `capacity` value, then it will utilize the small buffer optimization with an embedded capacity of that size.

Memory Usage

The `location` switch specifies whether the elements are managed at the front, back or middle of the capacity. `FRONT` provides vector-like behavior, with elements maintained at the lowest memory locations. `BACK` provides stack-like behavior (last-in on top), with elements maintained at the highest memory locations. `MIDDLE` supports the notion of a "shifty-vector" (a contiguous implementation of a deque), with elements floating in the middle of the capacity. The first element is placed in the middle of the capacity. When one end or the other would be exceeded by an insertion, the elements are recentered in the capacity if there is room (otherwise reallocation occurs if allowed—see below).

Reallocation occurs for dynamically allocated variable capacity sequences if an insertion is performed which would exceed the current capacity (as with vector). The growth of the capacity is governed by the `growth` mode which specifies whether the growth is `LINEAR` or `EXPONENTIAL`. The linear growth rate is specified by the `increment` value in elements. The exponential growth rate is specified by the `factor` value as a real number greater than 1.0.

The third `VECTOR` growth mode may or may not be needed: it is intended to provide implementation-defined growth behavior matching the behavior of `std::vector`, even if that behavior cannot be described exactly by either of the other growth modes. If `VECTOR` is provided, then it probably would also be the default growth mode (although perhaps this should be an implementation decision).

Removing elements from a sequence never causes reallocation or repositioning of elements (as with `std::vector`), but see `shrink_to_fit` below.

Operations

This sequence will support full access to both the front and the back. This is necessary because one of the most important design goals is the ability to change internal behavior of the sequence without changing client code. For more discussion about providing *O*(n) operations, see P0563.

The capacity operations `reserve` and `shrink_to_fit` are supported, and are quiet no-ops for fixed capacity sequences. For SBO sequences, `shrink_to_fit` has no effect if the elements are in the local buffer, but if the elements are in dynamic storage and the size is less than or equal to the local buffer size, the elements will be moved to local storage (and the dynamic allocation freed).

The `is_dynamic` test returns a bool indicating whether the elements are in dynamic storage. This is (mostly) equivalent to the `is_clumped` test described in P0274 (but the sense is reversed).

Errors

The behavior of the sequence when limits are exceeded, and ways to avoid various error states (e.g. `try_push_back`) will be the subject of considerable discussion and future proposals. P0843 deals with many of these issues.

Allocators

The subject of allocators has been addressed at some length for `inplace_vector`. (That discussion is an excellent example of the applicability of work on P0843 to this proposal.) The current consensus of that discussion is that allocators should be addressed later with a separate proposal. I strongly agree with this approach, and I suspect that allocator-awareness will be found to be a valid reason to create a separate entity (rather than adding it to the non-aware template).

My recommendation is that this sequence container be standardized initially without custom allocator support, with the expectation that the design process will be informed by the many years of experience with allocators in the Library. If we standardize this container, a proposal to add allocator support will no doubt shortly follow.

Other Designs

Include array

This sequence could include array-type behavior. I do not recommend this. While arrays do have contiguous memory, they are otherwise very different from variable-size sequences, and I think they are best addressed by a separate tool (`std::array`).

Use vector

A possible design approach for this sequence would be to treat it as "Vector 2.0" and try to expand `std::vector` to do everything. I am skeptical that there is a way to do this without either breaking compatibility with decades of existing usage or seriously compromising the design. I also feel strongly that we should take the opportunity to give this sequence a better name.

Another related approach would be to attempt to add this functionality to `std::vector` with custom allocators. This approach was considered and rejected by both P0843 and P0274, which see for details.

References

P0274R0: *Clump – A Vector-like Contiguous Sequence Container with Embedded Storage.* Liber.
P0563R0: *Vector Front Operations.* Talbot.
P0843R10: *inplace_vector.* Gadeschi, Doumler, Liber, Sankel.

Acknowledgements

Thanks to Timur Doumler, Howard Hinnant, and Nevin Liber for their time and encouragement to pursue this proposal. Thanks to Jonathan Wakely for introducing me to shifty-vector. Thanks to Wouter Vermaelen for suggesting adding control over the size type.