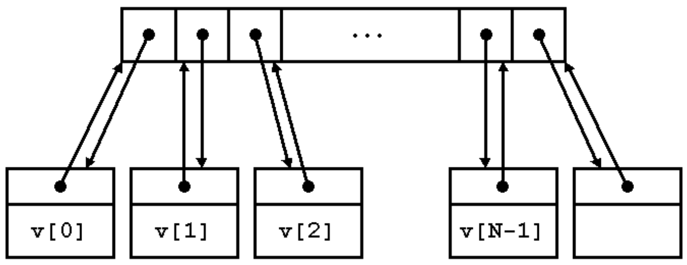
Non-standard containers

stable_vector

This useful, fully STL-compliant stable container designed by Joaquín M. López Muñoz is an hybrid between `vector` and `list`, providing most of the features of `vector` except element contiguity.

Extremely convenient as they are, `vector`s have a limitation that many novice C++ programmers frequently stumble upon: iterators and references to an element of an `vector` are invalidated when a preceding element is erased or when the vector expands and needs to migrate its internal storage to a wider memory region (i.e. when the required size exceeds the vector's capacity). We say then that `vector`s are unstable: by contrast, stable containers are those for which references and iterators to a given element remain valid as long as the element is not erased: examples of stable containers within the C++ standard library are `list` and the standard associative containers (`set`, `map`, etc.).

Sometimes stability is too precious a feature to live without, but one particular property of `vector`s, element contiguity, makes it impossible to add stability to this container. So, provided we sacrifice element contiguity, how much can a stable design approach the behavior of `vector` (random access iterators, amortized constant time end insertion/deletion, minimal memory overhead, etc.)? The following image describes the layout of a possible data structure upon which to base the design of a stable vector:



Each element is stored in its own separate node. All the nodes are referenced from a contiguous array of pointers, but also every node contains an "up" pointer referring back to the associated array cell. This up pointer is the key element to implementing stability and random accessibility:

Iterators point to the nodes rather than to the pointer array. This ensures stability, as it is only the pointer array that needs to be shifted or relocated upon insertion or deletion. Random access operations can be implemented by using the pointer array as a convenient intermediate zone. For instance, if the iterator it holds a node pointer `it.p` and we want to advance it by n positions, we simply do:

```
it.p = *(it.p->up+n);
```

That is, we go "up" to the pointer array, add n there and then go "down" to the resulting node.

General properties. `stable_vector` satisfies all the requirements of a container, a reversible container and a sequence and provides all the optional operations present in vector. Like vector, iterators are random access. `stable_vector` does not provide element contiguity; in exchange for this absence, the container is stable, i.e. references and iterators to an element of a `stable_vector` remain valid as long as the element is not erased, and an iterator that has been assigned the return value of end() always remain valid until the destruction of the associated `stable_vector`.

Operation complexity. The big-O complexities of `stable_vector` operations match exactly those of vector. In general, insertion/deletion is constant time at the end of the sequence and linear elsewhere. Unlike vector, `stable_vector` does not internally perform any value_type destruction, copy/move construction/assignment operations other than those exactly corresponding to the insertion of new elements or deletion of stored elements, which can sometimes compensate in terms of performance for the extra burden of doing more pointer manipulation and an additional allocation per element.

Exception safety. (according to Abrahams' terminology) As `stable_vector` does not internally copy/move elements around, some operations provide stronger exception safety guarantees than in vector:

Table 8.1. Exception safety

operation	**exception safety for** `vector<T>`	**exception safety for** `stable_vector<T>`
insert	strong unless copy/move construction/assignment of `T` throw (basic)	strong
erase	no-throw unless copy/move construction/assignment of `T` throw (basic)	no-throw

Memory overhead. The C++ standard does not specify requirements on memory consumption, but virtually any implementation of `vector` has the same behavior with respect to memory usage: the memory allocated by a `vector` v with n elements of type T is

$m_v = c \cdot e$,

where c is `v.capacity()` and e is `sizeof(T)`. c can be as low as n if the user has explicitly reserved the exact capacity required; otherwise, the average value c for a growing `vector` oscillates between 1.25·n and 1.5·n for typical resizing policies. For `stable_vector`, the memory usage is

$m_{sv} = (c + 1)p + (n + 1)(e + p)$,

where p is the size of a pointer. We have c + 1 and n + 1 rather than c and n because a dummy node is needed at the end of the sequence. If we call f the capacity to size ratio c/n and assume that n is large enough, we have that

$m_{sv}/m_v \simeq (fp + e + p)/fe$.

So, `stable_vector` uses less memory than `vector` only when e > p and the capacity to size ratio exceeds a given threshold:

$m_{sv} < m_v \leftrightarrow f > (e + p)/(e - p)$. (provided e > p)

This threshold approaches typical values of f below 1.5 when e > 5p; in a 32-bit architecture, when e > 20 bytes.

Summary. `stable_vector` is a drop-in replacement for `vector` providing stability of references and iterators, in exchange for missing element contiguity and also some performance and memory overhead. When the element objects are expensive to move around, the performance overhead can turn into a net performance gain for `stable_vector` if many middle insertions or deletions are performed or if resizing is very frequent. Similarly, if the elements are large there are situations when the memory used by `stable_vector` can actually be less than required by vector.

Note: Text and explanations taken from Joaquín's blog

flat_(multi)map/set associative containers

Using sorted vectors instead of tree-based associative containers is a well-known technique in C++ world. Matt Austern's classic article Why You Shouldn't Use set, and What You Should Use Instead (C++ Report 12:4, April 2000) was enlightening:

"Red-black trees aren't the only way to organize data that permits lookup in logarithmic time. One of the basic algorithms of computer science is binary search, which works by successively dividing a range in half. Binary search is log N and it doesn't require any fancy data structures, just a sorted collection of elements. (...) You can use whatever data structure is convenient, so long as it provides STL iterator; usually it's easiest to use a C array, or a vector."

"Both std::lower_bound and set::find take time proportional to log N, but the constants of proportionality are very different. Using g++ (...) it takes X seconds to perform a million lookups in a sorted vector<double> of a million elements, and

almost twice as long (...) using a set. Moreover, the set uses almost three times as much memory (48 million bytes) as the vector (16.8 million)."

"Using a sorted vector instead of a set gives you faster lookup and much faster iteration, but at the cost of slower insertion. Insertion into a set, using set::insert, is proportional to log N, but insertion into a sorted vector, (...) , is proportional to N. Whenever you insert something into a vector, vector::insert has to make room by shifting all of the elements that follow it. On average, if you're equally likely to insert a new element anywhere, you'll be shifting N/2 elements."

"It may sometimes be convenient to bundle all of this together into a small container adaptor. This class does not satisfy the requirements of a Standard Associative Container, since the complexity of insert is O(N) rather than O(log N), but otherwise it is almost a drop-in replacement for set."

Following Matt Austern's indications, Andrei Alexandrescu's Modern C++ Design showed `AssocVector`, a `std::map` drop-in replacement designed in his Loki library:

"It seems as if we're better off with a sorted vector. The disadvantages of a sorted vector are linear-time insertions and linear-time deletions (...). In exchange, a vector offers about twice the lookup speed and a much smaller working set (...). Loki saves the trouble of maintaining a sorted vector by hand by defining an AssocVector class template. AssocVector is a drop-in replacement for std::map (it supports the same set of member functions), implemented on top of std::vector. AssocVector differs from a map in the behavior of its erase functions (AssocVector::erase invalidates all iterators into the object) and in the complexity guarantees of insert and erase (linear as opposed to constant). "

Boost.Container `flat_[multi]map/set` containers are ordered, vector-like container based, associative containers following Austern's and Alexandrescu's guidelines. These ordered vector containers have also benefited with the addition of `move semantics` to C++11, speeding up insertion and erasure times considerably. Flat associative containers have the following attributes:

- Faster lookup than standard associative containers
- Much faster iteration than standard associative containers. Random-access iterators instead of bidirectional iterators.
- Less memory consumption for small objects (and for big objects if `shrink_to_fit` is used)
- Improved cache performance (data is stored in contiguous memory)
- Non-stable iterators (iterators are invalidated when inserting and erasing elements)
- Non-copyable and non-movable values types can't be stored
- Weaker exception safety than standard associative containers (copy/move constructors can throw when shifting values in erasures and insertions)
- Slower insertion and erasure than standard associative containers (specially for non-movable types)

devector

`devector` is a hybrid of the standard vector and deque containers originally written by Thaler Benedek. It offers cheap (amortized constant time) insertion at both the front and back ends, while also providing the regular features of `vector`, in particular the contiguous underlying memory.

Unlike `vector`, devector can have free capacity both before and after the elements. This enables efficient implementation of methods that modify the devector at the front. In general, `devector`'s available methods are a superset of those of `vector` with similar behaviour, barring a couple of iterator invalidation guarantees that differ.

The static size overhead for boost's devector is one extra `size_t` per container: Usually sizeof(devector) == 4*sizeof(T*).

There are different strategies when elements are to be inserted at one extreme of the container and there is no room for additional elements at that extreme. One simple strategy would be to reallocate a bigger buffer and move all elements to the new memory. However, this would lead to unbounded memory waste when elements are inserted predominantly on one extreme (e.g. pushed at one extreme and popped from the other, like a LIFO pattern).

To avoid unbounded memory waste, Boost.Container's `devector` uses a different strategy:

- If elements are inserted near a extreme and there is free space on that extreme, the insertion is performed without any additional data movement (only the elements between the insertion point and the extreme are moved.
- If elements are inserted near one extreme and the free space on that extreme is exhausted, all existing elements are relocated (moved) to the center of the internal memory buffer. This makes room in the exhausted extreme to insert more elements without allocating a new buffer.
- Potentially, the maximum number of possible relocations (movements) reusing the memory buffer are Olog(N), but that would lead non-amortized constant-time insertion at the extremes. In consequence, the number of relocations must be limited ('relocation limit') and a reallocation (allocation of a new memory buffer) will be performed if the load-factor of the container defined as (size()/length_of_buffer) surpasses the relocation limit (see Lars Greger Nordland Hagen's "Double-ended vector - is it useful?" article for more details.
- This approach offers a reasonable balance between a reasonable memory overhead and performance.

However, this strategy has also some downsides:

- Insertions at the extremes have no strong exception guarantee as data has to be move inside the existing buffer.
- Due to the memory relocation vs reallocation strategy explained above:
    - `capacity()` can no longer tell the maximum number of elements that the container can hold and, at the same time, the number of insertions to perform before a reallocation is performed. Depending on which extreme a insertion takes place, a reallocation might occur or not (maybe there is free capacity at that extreme)
- Instead of removing the `capacity()` member or renaming it to "`minimum_capacity()`", the definition has been changed to tell the **minimum**

number of elements that can be inserted without reallocating. This allows the typical pattern where:
  - If `reserve(n)` is called, `capacity() >= n`
  - If `capacity() == n` it is guaranteed that if `size() <= n` no reallocation will occur.
- However the usual container invariant where `size() <= capacity()` does not hold. `size()` can be bigger than `capacity()` because elements can be always inserted at an extreme with free capacity without reallocation.

slist

When the standard template library was designed, it contained a singly linked list called `slist`. Unfortunately, this container was not standardized and remained as an extension for many standard library implementations until C++11 introduced `forward_list`, which is a bit different from the the original SGI `slist`. According to SGI STL documentation:

"*An* `slist` *is a singly linked list: a list where each element is linked to the next element, but not to the previous element. That is, it is a Sequence that supports forward but not backward traversal, and (amortized) constant time insertion and removal of elements. Slists, like lists, have the important property that insertion and splicing do not invalidate iterators to list elements, and that even removal invalidates only the iterators that point to the elements that are removed. The ordering of iterators may be changed (that is, slist<T>::iterator might have a different predecessor or successor after a list operation than it did before), but the iterators themselves will not be invalidated or made to point to different elements unless that invalidation or mutation is explicit.*"

"*The main difference between* `slist` *and list is that list's iterators are bidirectional iterators, while slist's iterators are forward iterators. This means that* `slist` *is less versatile than list; frequently, however, bidirectional iterators are unnecessary. You should usually use* `slist` *unless you actually need the extra functionality of list, because singly linked lists are smaller and faster than double linked lists.*"

"*Important performance note: like every other Sequence,* `slist` *defines the member functions insert and erase. Using these member functions carelessly, however, can result in disastrously slow programs. The problem is that insert's first argument is an iterator pos, and that it inserts the new element(s) before pos. This means that insert must find the iterator just before pos; this is a constant-time operation for list, since list has bidirectional iterators, but for* `slist` *it must find that iterator by traversing the list from the beginning up to pos. In other words: insert and erase are slow operations anywhere but near the beginning of the slist.*"

"*Slist provides the member functions insert_after and erase_after, which are constant time operations: you should always use insert_after and erase_after whenever possible. If you find that insert_after and erase_after aren't adequate for your needs, and that you often need to use insert and erase in the middle of the list, then you should probably use list instead of slist.*"

Boost.Container updates the classic `slist` container with C++11 features like move semantics and placement insertion and implements it a bit differently than the standard C++ `forward_list`. `forward_list` has no `size()` method, so it's been designed to allow (or in practice, encourage) implementations without tracking list size with every insertion/erasure, allowing constant-time `splice_after(iterator, forward_list &, iterator, iterator)`-based list merging. On the other hand `slist` offers constant-time `size()` for those that don't care about linear-time `splice_after(iterator, slist &, iterator, iterator)` `size()` and offers an additional `splice_after(iterator, slist &, iterator, iterator, size_type)` method that can speed up `slist` merging when the programmer already knows the size. `slist` and `forward_list` are therefore complementary.

static_vector

`static_vector` is an hybrid between `vector` and `array`: like `vector`, it's a sequence container with contiguous storage that can change in size, along with the static allocation, low overhead, and fixed capacity of `array`. `static_vector` is based on Adam Wulkiewicz and Andrew Hundt's high-performance varray class.

The number of elements in a `static_vector` may vary dynamically up to a fixed capacity because elements are stored within the object itself similarly to an array. However, objects are initialized as they are inserted into `static_vector` unlike C arrays or `std::array` which must construct all elements on instantiation. The behavior of `static_vector` enables the use of statically allocated elements in cases with complex object lifetime requirements that would otherwise not be trivially possible. Some other properties:

- Random access to elements
- Constant time insertion and removal of elements at the end
- Linear time insertion and removal of elements at the beginning or in the middle.

`static_vector` is well suited for use in a buffer, the internal implementation of other classes, or use cases where there is a fixed limit to the number of elements that must be stored. Embedded and realtime applications where allocation either may not be available or acceptable are a particular case where `static_vector` can be beneficial.

small_vector

`small_vector` is a vector-like container optimized for the case when it contains few elements. It contains some preallocated elements in-place, which allows it to avoid the use of dynamic storage allocation when the actual number of elements is below that preallocated threshold. `small_vector` is inspired by LLVM's `SmallVector` container. Unlike `static_vector`, `small_vector`'s capacity can grow beyond the initial preallocated capacity.

`small_vector<T, N, Allocator>` is convertible to `small_vector_base<T, Allocator>`, a type that is independent from the preallocated element count, allowing client code that does not need to be templated on that N argument. `small_vector` inherits all `vector`'s member functions so it supports all standard features like emplacement, stateful allocators, etc.