Document number: **P0563R0**
Date: 2017-02-05
Audience: Library Evolution Working Group
Reply to: **Alan Talbot**
cpp@alantalbot.com

Vector Front Operations

Once upon a time, a long, long time ago, the wise men and women who created the Standard Template Library sought to save us, the working programmers of the world, from the dreaded fate of inefficiency. They did this by holding back certain incantations that were thought so dangerous that no one but an elder wizard should even consider using them. But the world has changed, and now young people everywhere have years of wizard training, so perhaps it is time to reveal these dark spells. I refer of course to **push_front** and **pop_front** for **vector**.

Pushing onto the front of a vector was once expensive. It's O(n)—how could this ever be good? Well, computers have changed—a lot. With modern architectures, locality of reference is far more important than big-O complexity when the container size is small (i.e. fits in cache). Today, sliding a relatively small amount of memory around is very fast, while allocating memory on the heap is very slow. And accessing data that is scattered around the heap is much, much slower than accessing data that is contiguous. For small vectors, the O(n) operations that were once slow are actually much faster than using a more complicated container. (Note: this argument works in reverse with **forward_list**, for which O(n) back operations are especially expensive in cached environments.)

For large vectors the O(n) property still dominates, but I find that a *very* common use case is to create small lists and use them in high frequency and/or high volume applications. To simulate this, I did a simple test on a modest workstation machine (3.5 GHz, 4 cores, 8 logical processors) using one of the leading compilers and operating systems. To create a somewhat realistic scenario, I put a fairly large number of small objects in a **map** (a table pattern that I use often). Each object contained a small list of integers which I inserted one at a time at the front of the list. I then removed them, again one at a time from the front. I ran tests implementing the list with a standard **list**, **deque** and **vector**. In the **vector** case I reserved the correct size in advance. I tried two different size scenarios: 10M objects with 10 ints, and 1M objects with 100 ints.

	Container	Time (s)	Memory (GB)
10M/10	List	8.1	4.34
	Deque	3.7	3.05
	Vector	2.7	1.29
1M/100	List	6.4	3.33
	Deque	1.7	1.21
	Vector	2.6	0.50

The times listed are the time spent adding and removing the ints, as timed by the operating system's microsecond-precision interval timer. The time spent building and destroying the map holding the objects was not included. The memory measured was the peak usage—the working set after loading the ints.

In the 10 element case, the **vector** beat the **deque** handily in speed and used less than half the memory. In the 100 element case, while the **deque** beat the **vector** in speed, the **vector** again used less than half the memory. Unsurprisingly, the **list** was far behind in both measurements.

The **vector** container is meant to be the go-to container for most situations, so it should not be a second class citizen. The feature sets of the sequence containers should be consistent—this is especially important in generic contexts—and the choice of container ought to be driven by the requirements of the situation, not the convenience of the API. I therefore propose adding **push_front**, **pop_front**, and **emplace_front** to **vector**, making it consistent with **list** and **deque**, and more supportive of modern computers.

Proposed Wording

23.2.3 Sequence containers **[sequence.reqmts]**

¶15 Table 88

emplace_front, container column:

`deque, forward_list, list,` **`vector`**

push_front (both t and rv versions), container column:

`deque, forward_list, list,` **`vector`**

pop_front, container column:

`deque, forward_list, list,` **`vector`**

23.3.11.1 Class template vector overview **[vector.overview]**

¶2 // 23.3.11.5, modifiers, add:

```
template <class... Args> reference emplace_front(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
void pop_front();
```

23.3.11.5 vector modifiers **[vector.modifiers]**

Add before ¶1:

```
template <class... Args> reference emplace_front(Args&&... args);
void push_front(const T& x);
void push_front(T&& x);
```

Add before ¶3:

```
void pop_front();
```

23.3.12 Class vector<bool> **[vector.bool]**

¶1 // modifiers, add:

```
template <class... Args> reference emplace_front(Args&&... args);
void push_front(const bool& x);
void pop_front();
```

23.6.4.1 queue definition **[queue.defn]**

¶1

> Any sequence container supporting operations `front()`, `back()`, `push_back()` and `pop_front()` can be used to instantiate `queue`. In particular, **`vector (23.3.11),`** `list` (23.3.10) and `deque` (23.3.8) can be used.