

*Document number:* **P0084R2**  
*Date:* 2016-06-23  
*Audience:* Library Evolution Working Group, Library Working Group  
*Reply to:* **Alan Talbot**  
cpp@alantalbot.com

---

## Emplace Return Type (Revision 2)

---

I often find myself wanting to create an element of a container using **emplace\_front** or **emplace\_back**, and then access that element, either to modify it further or simply to use it. So I find myself writing code like this:

```
my_container.emplace_back(...);  
my_container.back().do_something(...);
```

Or perhaps:

```
my_container.emplace_back(...);  
do_something_else(my_container.back());
```

Quite a common specific case is where I need to construct an object before I have all the information necessary to put it into its final state, such as when I'm reading it from a file:

```
my_container.emplace_back();           // Default construct.  
my_container.back().read(file_stream); // Read the object.
```

This happens often enough that I tend to write little templates that call some version of **emplace** and return **back**, which seems rather unnecessary to me. I believe the **emplace\_front** and **emplace\_back** functions should return a non-const reference to the newly created element, in keeping with the current Standard Library trend of returning useful information when practical. It was an oversight (on my part) in the original **emplace** proposal that they do not.

---

### Push Functions

A similar argument could be made for **push\_front** and **push\_back**, but I believe that the argument is weaker because if you are using one of the push functions, you typically already have a complete object in hand. While it is true that situations can arise where you want access to the newly formed element, by nature they are less frequent than with **emplace**. Furthermore, if you want that behavior you can always use the **emplace** version. Therefore I am not proposing changes to the push functions.

---

### Changes in Revision 1

This paper updates P0084R0 as follows:

- Added Table 108 modifications.
- Added a discussion about ABI breakage.
- Added a discussion about performance.

---

## ABI Breakage

When this paper was discussed by the LEWG in Kona (October 2015), there was general approval of the concept, but there was a concern raised by Jonathan Wakely that it would break ABI for C++17. This resulted in the paper being rejected. I have discussed this with Jon and he believes he was mistaken—this change is to a template and return types are part of the mangled name for templates, therefore this will not break ABI.

---

## Performance

A question was raised in Kona about the performance effects of this change. Howard Hinnant did a small test with one compiler and there was no code generated for a return type if the return value was not used. I take this as good evidence that there will be no performance implications.

---

## Proposed Wording

### 23.2.3 Sequence containers

[sequence.reqmts]

¶16 Table 108

#### emplace\_front

Return type: **void reference**.Append to operational semantics: *Returns: a.front()*.

#### emplace\_back

Return type: **void reference**.Append to operational semantics: *Returns: a.back()*.

### 23.3.8.1 Class template deque overview

[deque.overview]

¶2 // 23.3.8.4, modifiers:

template <class... Args> **void reference** emplace\_front(Args&&... args);template <class... Args> **void reference** emplace\_back(Args&&... args);

### 23.3.8.4 deque modifiers

[deque.modifiers]

template <class... Args> **void reference** emplace\_front(Args&&... args);template <class... Args> **void reference** emplace\_back(Args&&... args);

### 23.3.9.1 Class template forward\_list overview

[forwardlist.overview]

¶3 23.3.9.5, modifiers:

template <class... Args> **void reference** emplace\_front(Args&&... args);

### 23.3.9.5 forward\_list modifiers

[forwardlist.modifiers]

¶2

template <class... Args> **void reference** emplace\_front(Args&&... args);

### 23.3.10.1 Class template list overview

[list.overview]

¶2 // 23.3.10.4, modifiers:

template <class... Args> **void reference** emplace\_front(Args&&... args);template <class... Args> **void reference** emplace\_back(Args&&... args);*[Editorial note: pop\_front should follow the second push\_front.]*

**23.3.10.4 list modifiers****[list.modifiers]**

```
template <class... Args> void reference emplace_front(Args&&... args);
template <class... Args> void reference emplace_back(Args&&... args);
```

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

¶2 23.3.11.5, modifiers:

```
template <class... Args> void reference emplace_back(Args&&... args);
```

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

```
template <class... Args> void reference emplace_back(Args&&... args);
```

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

¶1

```
template <class... Args> void reference emplace_back(Args&&... args);
```

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

¶1

```
template <class... Args>
void reference emplace(Args&&... args)
{ return c.emplace_back(std::forward<Args>(args)...); }
```

**23.6.6.1 stack definition****[stack.defn]**

```
template <class... Args>
void reference emplace(Args&&... args)
{ return c.emplace_back(std::forward<Args>(args)...); }
```

---

**Acknowledgements**

Thanks to Jonathan Wakely and Howard Hinnant for researching the issues of possible concern.