

Document number: **P0084R0**
Date: 2015-09-24
Project: Programming Language C++
Reference: N4527
Reply to: **Alan Talbot**
cpp@alantalbot.com

Emplace Return Type

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. This is in keeping with the current trend of returning useful information when practical. It was an oversight 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.

Proposed Wording

The standard does not state what is actually returned from **front** or **back**, presumably with the assumption that it is obvious, so I have not proposed adding such language for **emplace_front** and **emplace_back**. If there is any concern about ambiguity, I can add language stating that they return a reference to the newly added element.

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

¶16 Table 100 - Change return type for `emplace_front` and `emplace_back` from `void` to `reference`.

23.3.3.1 Class template deque overview**[deque.overview]**

¶2 // 23.3.3.4, modifiers:

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

23.3.3.4 deque modifiers**[deque.modifiers]**

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

23.3.4.1 Class template forward_list overview**[forwardlist.overview]**

¶3 23.3.4.5, modifiers:

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

23.3.4.5 forward_list modifiers**[forwardlist.modifiers]**

¶2

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

23.3.5.1 Class template list overview**[list.overview]**

¶2 // 23.3.5.4, 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`.]

23.3.5.4 list modifiers**[list.modifiers]**

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

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

¶2 23.3.6.5, modifiers:

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

23.3.6.5 vector modifiers**[vector.modifiers]**

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

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

¶1

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

23.6.3.1 queue definition**[queue.defn]**

¶1

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

23.6.5.2 stack definition**[stack.defn]**

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