Proposed Wording for Placement Insert (Revision 1)

Document number:	N2680 = 08-0190
Date:	2008-06-13
Project:	Programming Language C++

References:	N2345 = 07-0205
	N2606 = 08-0116
	N2642 = 08-0152

Reply to:	Alan Talbot alan.talbot@teleatlas.com Tele Atlas North America 11 Lafayette St Lebanon NH 03766 USA

Abstract

This paper provides new proposed wording for the addition of placement insert operations to the standard containers, and addresses several issues that have been raised. Readers unfamiliar with the placement insert operations are encouraged to read the latest version of the full proposal (N2345). Note that the proposed wording herein is based on the Working Draft (N2606) that includes the full proposal.

Summary of Motivation

The motivation for placement insert is that containers—especially node-based containers—are very useful for the storage of heavy objects. In some environments efficiency is very important, but there is in general no way to put elements into containers without copying them. A heavy object may store its data directly, in which case move semantics will not improve copy performance. Furthermore, when efficiency is critical the solution cannot depend on compiler optimizations since they are optional and may not occur where they are most needed.

Placement insertion lets us create an element once, in the container where we want it, and never have to move it or copy it. It does so in a simple and direct way by introducing new variadic functions that take arguments that are passed on to the element's constructor using variadic templates and perfect forwarding.

Pair Issues

Questions have come up as to why pair requires a variadic constructor, and why only on the second element. (What makes that special?) The answer to the first question is that map is defined to have a value type of (approximately) pair<key_type, mapped_type>. In order to in-place construct such a type, the pair constructor would in theory have to take arguments to construct both of its members.

However, variadic templates provide no way of supplying two argument lists (such a facility would require a new syntax at the call site). But since the key_type of a map is in practice much less likely to require in-place construction, I believe that the common (albeit asymmetrical) use case of a copy (or move) constructed key_type with an in-place constructed mapped_type is well worth supporting.

To provide this, pair requires a variadic constructor to pass arguments through to the second element, hence the seemingly strange asymmetrical signature. This is not expected to be commonly used other than by map. Indeed, if map were not defined in terms of pair there would be no need for this constructor.

Note that this is not related to the variadic construction of tuples—that is a different and unrelated use of variadic templates.

There is also a zero problem with pair—see below.

Because the problems with pair are not limited to the changes introduced by placement insert, and because they are somewhat complicated to solve, the LWG decided to defer fixing pair to a separate paper.

Push_back Issues

The fundamental problem has to do with perception of the meaning of push_back and the meaning of emplace. My approach to the emplace issue has been based on the premise that if we were redesigning the Library from scratch, there would be no emplace. Insert would work whether you gave it an object or constructor arguments for the object. The same would hold for push_back and push_front, since they are simply conveniences and (perhaps) optimizations of insert. However, we are not designing the Library from scratch, so the prevailing understanding of the meaning of existing members must be preserved. More on this later.

The pedagogical problem

Concerns were raised on the Library reflector that push_back becomes hard to teach if its signature is (only) a variadic template. Further confusion arose over the meaning of push_back when called with multiple parameters. Was:

```
list<something> l;
l.push_back(a, b, c);
```

meant to mean:

```
l.push_back(a);
l.push_back(b);
l.push_back(c);
```

or was it meant to mean:

```
l.push_back(something(a, b, c));
```

The answer is the second interpretation, and this is of course easy to document, but the problem is that there was confusion even among experts. The concern is that this confusion would likely be worse for those learning the language.

The explicit problem

Questions have also been raised as to whether a variadic in-place construction call (e.g. push_back, emplace) would honor explicit (that is, fail if the targeted constructor were explicit). In fact it will not fail since it is not (at least technically) violating explicitness:

```
struct A {};

struct B {
   explicit B(A);
};

void foo(B) {}

A a;
foo(a);    // Error: explicit constructor prevents implicit conversion
foo(B(a)); // OK: constructor is called explicitly

list<B> lb;
lb.push_back(a);    // 1 - OK: constructor is called explicitly by push_back
lb.push_back(B(a)); // 2 - OK: constructor is called explicitly at call site
```

Note that the variadic push_back has no more license to convert things than the original function. You can't push arguments (1) that would not be legal as direct constructor arguments (2).

Chris Jefferson pointed out: "In particular I often see people trying to push '1' into a vector<vector<int>>, when they meant to either push it onto a single vector, or wanted to push back a new vector which contains 1. At the moment this fails, because the size constructor to vector is explicit, as we assume people would not want to implicitly turn an integer into a vector. Now this would be silently turned into an emplace, pushing back a new vector of length 1."

This same issue exists for any variadic construction function, so it affects emplace as well. I do not believe this is a problem for emplace because the purpose of emplace is to construct an object of (in this case) type B by passing something from which B can be constructed.

The argument has more weight with push_back however, because it is an existing function with a long history of being understood in a particular way. Changing formerly error-producing behavior into legal and possibly surprising behavior could easily be construed as breaking something.

By the way, looking again at Chris's example, with the prior proposal wording you could do the following:

```
vector<vector<int>> vv;
vv.push_back();
vv.back().push_back(1);
```

which to do efficiently in C++03 would require:

```
vector<vector<int>> vv;
vv.resize(vv.size() + 1);
vv.back().push_back(1);
```

This works fine, but it seems more confusing and harder to teach than the variadic push_back solution.

The zero problem (LWG 767)

The following code is legal in C++03:

```
vector<int*> v;
v.push_back(0);
```

But under the prior version of this proposal the following occurs:

```
vector<int*> v;
v.push_back();        // OK: first element is null pointer
v.push_back(nullptr); // OK: first element is null pointer
v.push_back(0);       // Error: int* cannot be initialize with int
```

What happens here is that the magic nature of 0 is lost because the only signature of push_back is the template, and the type is deduced to be int. This problem can be solved with a combination of restoring the original signature and constraining the template as follows:

```
void push_back(const T& x);
template <class... Args>
   requires Constructible<T, Args&&...>
   void push_back(Args&&... args);
```

The same problem occurs with pair constructor templates even without the emplace variadic signature:

```
template<class U, class V> pair(U&& x, V&& y); // (One of several constructors)
std::pair<char*, char*> p(0,0); // Error: int* cannot be initialize with int
```

Again, this can be solved with appropriate constraints and non-template overloads.

Emplace Overloads (LWG 763)

The prior proposal provided two overloads for emplace on associative containers that match the first two insert overloads. One provides construction arguments for a value type and the other a hint and construction arguments. This can make the signatures ambiguous if the first construction argument happens to be of type const_iterator into the same type of container. Because these are templates, it will not be literally ambiguous, but the one that gets called may not be the one you had in mind. For instance, if you have a set that has a value type that is constructible with a const_iterator:

```
class bar {
   bar();
   bar(set<bar>::const_iterator);
};
set<bar> s;
set<bar>::const_iterator i = something();
s.emplace(i); // Oops: effect is insert(i, bar())
```

This calls the hint version, which is probably not what you wanted. While non-pathological examples of this case are probably extremely rare, it is a nasty little problem that would be rather difficult to anticipate or debug if you ran into it.

There is a simple work around:

```
s.emplace(s.begin(), i);
```

In Bellevue I recommended this as the solution, but on reflection I see that it has two problems: 1) you have to have anticipated the problem to know you need to work around it, and 2) using an arbitrary hint like begin() is likely to cause poor performance (and performance is the whole reason for emplace).

Container adaptors (LWG 756)

The original proposal omitted container adaptors (23.2.5). This was an oversight on my part; thanks to Paolo Carlini for pointing it out. The wording below includes emplace versions of push for the adaptors.

Solutions

Since we are not rewriting the Library from scratch, we have to respect backward compatibility in two ways: technical and psychological. Various solutions have been suggested to each of the problems discussed above, and I believe that the technical problems are mostly or entirely solvable. But these technical solutions do not address the psychological problems. Even if technical problems are solved, I do not believe that we should ship a standard that creates significant psychological problems for users comfortable with the C++03 Library.

The push_back problems in particular are mostly psychological. There are reasonable technical solutions to the technical problems. The psychological problem is that many uses of containers are for simple types which do not have value constructors other than a copy constructor and do not exhibit inefficiency when constructed and copied unnecessarily. These types lead to a way of thinking about containers and using their functions makes the behavior of the new overloads rather surprising.

I would like to discuss two approaches to solving these problems. One is rather conservative, but also very straightforward. The other is very slick for the user, but may prove too difficult (or too expensive in some way).

The Conservative Approach

The simple solution to almost all of the concerns is to have separate names for all functions. For instance for list:

```
template<typename... Args> void emplace_front(Args&&... args);
template<typename... Args> void emplace_back(Args&&... args);
template<typename... Args> iterator emplace(const_iterator position, Args&&... args);
```

and for set:

```
template<typename... Args>
   pair<iterator, bool> emplace(Args&&... args);
template<typename... Args>
   iterator emplace_hint(const_iterator position, Args&&... args);
```

Now the pedagogical problem goes away because these are new separate functions and the containers can be taught without reference to them.

The explicit problem becomes less of an issue because the only time you are constructing an object is with some form of emplace, which is specifically designed to do just that. Explicit is meant to prevent *accidental construction* of an object of an *unexpected* type, but here the construction is not accidental and the type is precisely what is expected.

The zero problem goes away because we can dictate (for new functions) that nullptr is the only legal way to specify a non-specific null pointer type. 0 becomes just another int.

The ambiguity of overloads goes away because there is a different name for the hint version. This is probably something that would be done for insert as well if we were redesigning the Library, in keeping with the philosophy that over-overloading is not a good idea.

There is a reasonable argument that it is not necessary to provide emplace_back since it is typically simply a convenience function that calls emplace(end(), ...). But although I can see this point and agree with it in principle, I am not quite comfortable with leaving it out. The reason is that I believe that normal use of containers should be with these new functions, not insert and push_back. The old functions are made obsolete by emplace. If we were starting over, we would provide this functionality in insert and push_back in the first place. Why not? Given proper use of nullptr and separate names for the hint versions, there would be no reason not to.

The Fancy Approach

Pablo Halpern, Peter Dimov and others had an exchange on the reflector that suggested another, different approach. The final solution (as proposed by Peter) would look something like this in use:

```
list<something> l;
l.push_back(a);             // Copy construct an element from a
l.push_back(emplace(b, c)); // In-place construct an element from b,c
l.insert(p, a);             // Copy construct an element from a
l.insert(p, emplace(b, c)); // In-place construct an element from b,c
```

This has several excellent properties. One is that it does not require any new named functions. Another is that the overloads would not ever be ambiguous. A third is that an overload could be provided that would allow the in-place construction of the key value in a map. And best of all, it is very simple and easy to use—just add emplace wherever you want good performance. Other things in the library could also use this approach.

The only problem is that the implementation would be pretty tricky, involving some fancy metaprogramming with some sort of tuple. And it would have to avoid adding overhead in the passing of the parameters to the constructor. In fact, I am not sure if it is possible to implement this in practice (without some sort of language support). The consensus of the LWG was not to pursue this approach.

Proposed Wording

20.6.5 The default allocator [default.allocator]

The change here is to remove the original signature, leaving only the new variadic version. This is the way it was in the final approved paper (N2345). There may have been an issue that put the original version back, but I could not find one.

Remove:

```
void construct(pointer p, const T& val);
```

20.6.5.1 allocator members [allocator.members]

Remove:

```
void construct(pointer p, const_reference val);
```

11 *Effects:* `::new((void *)p) T(val)`

23.1 Container requirements

23.1.1 Sequences [sequence.reqmts]

Table 89: Optional sequence container operations

Change:

```
a.push_front(args)
a.push_back(args)
```

to:

```
a.emplace_front(args)
a.emplace_back(args)
```

Add:

expression	return type	assertion/note pre/post-condition	container
`a.push_front(t)`	`void`	`a.insert(a.begin(),t)` `Requires:ConstructibleAsElement<A, T, T> and T shall be CopyAssignable.`	`list, deque`
`a.push_front(rv)`	`void`	`a.insert(a.begin(),t)` `Requires:ConstructibleAsElement<A, T, T&&> and T shall be MoveAssignable.`	`list, deque`
`a.push_back(t)`	`void`	`a.insert(a.end(),t)` `Requires:ConstructibleAsElement<A, T, T> and T shall be CopyAssignable.`	`vector, list, deque, basic_string`
`a.push_back(rv)`	`void`	`a.insert(a.end(),t)` `Requires:ConstructibleAsElement<A, T, T&&> and T shall be MoveAssignable.`	`vector, list, deque, basic_string`

23.1.2 Associative containers [associative.reqmts]

Table 94: Associative container requirements (in addition to container)

Change:

```
a.emplace(p,args)
```

to:

```
a.emplace_hint(p,args)
```

23.1.2 Unordered associative containers [unord.req]

Table 96: Unordered associative container requirements (in addition to container)

Change:

```
a.emplace(p,args)
```

to:

```
a.emplace_hint(p,args)
```

23.2 Sequence containers [sequences]

23.2.2 Class template deque [deque]

Change in paragraph 2 – class deque:

```
template <class... Args> void push_front(Args&&... args);
template <class... Args> void push_back(Args&&... args);
```

to:

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

Add to paragraph 2 – class deque:

```
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

23.2.2.3 deque modifiers [deque.modifiers]

Change:

```
template <class... Args> void push_front(Args&&... args);
template <class... Args> void push_back(Args&&... args);
```

to:

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

Add:

```
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

23.2.3 Class template forward_list [forwardlist]

Change in paragraph 2 – class forward_list:

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

to:

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

Add to paragraph 2 – class forward_list:

```
void push_front(const T& x);
void push_front(T&& x);
```

23.2.3.4 forward_list modifiers [forwardlist.modifiers]

Change at paragraph 2:

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

to:

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

Add after paragraph 2:

```
void push_front(const T& x);
void push_front(T&& x);
```

3 *Effects:* Inserts a copy of x at the beginning of the list.

23.2.4 Class template list [list]

Change in paragraph 2 – class list:

```
template <class... Args> void push_front(Args&&... args);
template <class... Args> void push_back(Args&&... args);
```

to:

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

Add to paragraph 2 – class list:

```
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

23.2.4.3 list modifiers [list.modifiers]

Change:

```
template <class... Args> void push_front(Args&&... args);
template <class... Args> void push_back(Args&&... args);
```

to:

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

Add:

```
void push_front(const T& x);
void push_front(T&& x);
void push_back(const T& x);
void push_back(T&& x);
```

23.2.5 Container adaptors [container.adaptors]

23.2.5.1.1 queue definition [queue.defn]

Add to class queue:

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

23.2.5.2 Class template priority_queue [priority.queue]

Add to paragraph 1, class priority_queue:

```
template<typename... Args> void emplace(Args&&... args);
```

23.2.5.2.2 priority_queue members [priqueue.members]

Add new paragraph after 2:

```
template<typename... Args> void emplace(Args&&... args);
```

3 Effects:

```
c.emplace_back(std::forward<Args>(args)...);
push_heap(c.begin(), c.end(), comp);
```

23.2.5.3.1 stack definition [stack.defn]

Add to class stack:

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

23.2.6 Class template vector [vector]

Change in paragraph 2 – class vector:

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

to:

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

Add to paragraph 2 – class vector:

```
void push_back(const T& x);
void push_back(T&& x);
```

23.2.6.4 vector modifiers [vector.modifiers]

Change:

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

to:

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

Add:

```
void push_back(const T& x);
void push_back(T&& x);
```

23.3.1 Class template map [map]

Change in paragraph 2 – class map:

```
template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
```

to:

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
```

23.3.2 Class template multimap [multimap]

Change in paragraph 2 – class map:

```
template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
```

to:

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
```

23.3.3 Class template set [set]

Change in paragraph 2 – class map:

```
template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
```

to:

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
```

23.3.4 Class template multiset [multiset]

Change in paragraph 2 – class map:

```
template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
```

to:

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
```

23.4 Unordered associative containers [unord]

23.4.1 Class template unordered_map [unord.map]

Change in paragraph 3 – class unordered_map:

```
template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
```

to:

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
```

23.4.2 Class template unordered_multimap [unord.multimap]

Change in paragraph 3 – class unordered_map:

```
template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
```

to:

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
```

23.4.3 Class template unordered_set [unord.set]

Change in paragraph 3 – class unordered_map:

```
template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
```

to:

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
```

23.4.4 Class template unordered_multiset [unord.multiset]

Change in paragraph 3 – class unordered_map:

```
template <class... Args>
iterator emplace(const_iterator position, Args&&... args);
```

to:

```
template <class... Args>
iterator emplace_hint(const_iterator position, Args&&... args);
```

Acknowledgements

The following people asked questions, provided answers, or made suggestions on the Library reflector and in other conversations concerning this proposal. I have tried to represent all of them correctly and fairly. If I have failed to do so, or left someone off the list, my apologies! Thank you all *very* much for your time and thought on this proposal.

Matt Austern
Paolo Carlini
Beman Dawes
Peter Dimov
Gabriel Dos Reis
Doug Gregor
Pablo Halpern
Howard Hinnant
Chris Jefferson
Loïc Joly
Jens Maurer
Thorsten Ottosen
Sylvain Pion
P. J. Plauger
Bjarne Stroustrup
David Vandevoorde