Placement Insert for Containers (Revision 2)

Document number: N2345 = 07-0205
Date: 2007-07-18
Project: Programming Language C++
Reference: N2315 = 07-0175
Reply to: Alan Talbot
alan.talbot@teleatlas.com
Tele Atlas North America
11 Lafayette St
Lebanon NH 03766 USA

Abstract

This paper proposes the addition of "placement insert" operations to the standard containers. The benefits are improved performance without compromising design, and the ability to put non-CopyConstructible and even non-MoveConstructible objects into containers. I will motivate this with an example from my work, and discuss the tricks I use to get around the problem in the current language. I will then propose several solutions and recommend the one I believe is clearly superior.

Motivation

In my geographical work I often use Standard Library containers to store large numbers of moderate sized objects that are non-trivial to copy. For example, I may represent a stretch of road (known in the trade as an "edge") with an object that has quite a bit of embedded data and also owns several dynamically allocated objects and arrays.

Many of the standard containers store objects on the heap and do not move them. This stability is very useful and also very efficient since no copying is required. I can construct objects once and then refer to them for as long as the container exists. Even non-node-based containers can be used this way under certain circumstances.

However, the interface to these containers requires that each object be constructed and then copied. This is expensive, so I find myself using awkward idioms that avoid "real" construction (to avoid copying of dynamically allocated components) and incomplete default construction (since such construction is wasted).

What I would like to do is this:

```
// Data source - could be a simple struct or could be a stream of some sort.
class widget_data {};

// The object I wish to store in a map.
class widget {
public:

    // No default constructor - this class is not meant to be default constructed.
    widget(const widget_data&) { ... } // Load all real data – full construction.
    ~widget() { ... } // Release dynamic data as necessary.

private:

    // Private copying - this class is not meant to be copied.
    widget(const widget&) {}
    widget& operator=(const widget_data&) {}

    // Embedded data here.
    // Dynamically allocated data here.
};
```

Widget is now used like this:

```
map<long, widget> m;

// For each record in my dataset, do the following:

long id;             // The key gets set somehow.
widget_data wd;      // The data source gets loaded somehow.
m.insert(make_pair(id, widget(wd)));
```

The first problem I run into is that this won't compile because my copy constructor is private, so I deteriorate my design a bit and make it public. Now I find that after I construct my widget, the library copies it and throws away the original (more than once). This is potentially expensive, and in some RAII cases might be unacceptable (if constructing the widget launches a rocket, for example).

An obvious solution is to allocate the objects myself and put pointers into the container. This is a nuisance, dangerous unless you do it right (with smart pointers), and kind of embarrassing (C++ gets enough grief for its use of pointers). And there is a more serious problem: it has a significant memory cost. In my work I am always tight for memory, so I'm not willing to trade memory for speed unless the speed improvement is very large.

So what I end up doing is what could be considered a trick. I use trivial default construction (to avoid useless or ill-advised initialization), the `operator[]` function (because it makes fewer copies on the implementation I use) , and finally assignment. This makes all of the unnecessary copies as trivial as possible and results in good performance, but it has a big impact on my design and it causes people to scratch their heads when they first see my code.

Here's what it looks like:

```
class widget {
public:

    // No real data constructor. Bad design.
    widget() {} // Trivial default constructor. Bad design.
    widget(const widget&) { ... } // Public copy constructor. Bad design.
    ~widget() { ... } // Release dynamic data as necessary.

    // Public copy assignment. Bad design.
    // This loads the real data, filling the role of an appropriate constructor.
    widget& operator=(const widget_data&) { ... }

private:

    // Embedded data here.
    // Dynamically allocated data here.
};
```

Widget is now used like this:

```
map<long, widget> m;
long id;             // The key gets set somehow.
widget_data wd;      // The data source gets loaded somehow.
m[id] = wd;
```

With the implementation I am using, this yields a default construct and two empty copy constructs, the assignment, and of course three destructs. If I use insert instead:

```
m.insert(make_pair(id, widget())).first->second = wd;
```

I get a default construct and *three* empty copy constructs, the assignment, and *four* destructs. (The additional copy is due to `make_pair`.)

This trick works in cases where the default construction and "empty" copies are pretty cheap, but it will not work well if the class has large embedded data. Furthermore, I have had to do something that I consider tricky, and I've compromised my design.

Another major drawback that my trick does *not* solve is that contained objects must still be copy constructible. This prohibits putting things like streams into containers, which is annoying and embarrassing.

What I really want is some way of constructing my object once, in place. Since the object will be instantiated on the heap and never moved, this should not be difficult. Unfortunately the interface does not offer a way to do this.

Solutions

General Comments

There are several possible solutions to this problem depending on which new language features one uses. I will discuss four of these, in increasing order of desirability and language support. All but the first of these solutions involve new member functions for containers. I also look at each container and discuss some specific considerations.

Move Semantics Alone

Move semantics will allow the copy construction done by `insert` to be replaced by move construction. This is a natural consequence of adapting the Library to rvalue references and move semantics. This will make my `operator[]` trick unnecessary because my class could define a move constructor which would be as fast as an "empty" copy.

However, this does not address the case where the object itself is very large, nor does it make it possible to put non-movable objects into containers. It does not really fix the design problem either. Certainly there are many cases where moving would be acceptable while copying would not (streams for example). But I think that there are cases where the object should not be copyable and should also not be movable, and making it movable to improve performance is a bit of a kludge at best.

Simple Placement Insert

A placement insert function would solve all of these problems. There are several ways to define such a function (which I call *emplace*). The simplest might work like this (given the first definition of widget above):

```
map<long, widget> m;
long id;            // The key gets set somehow.
widget_data wd;     // The data source gets loaded somehow.
pair<map<long, widget>::iterator, widget*> p = m.emplace(id);
if (p.second)
    new(p.second) widget(wd);
```

This only requires a single construction (and single destruction). All the unnecessary overhead is eliminated. The problem here is that if the insert succeeds, the returned iterator is pointing to an as-yet-unconstructed object. In fact, the more general issue is that after the call to `emplace`, the map is in a well formed state but one of its contained objects is not. This means that we must count on the programmer to do the right thing.

For a set the value is the key, so it requires a fully constructed object to do the lookup. This simple approach to placement insert would have to be done in two steps: first the object is constructed in a place provided by the implementation, then the lookup and linking are done to that object:

```
set<widget> s;
widget_data wd;     // The data source gets loaded somehow.
if (widget* p = s.next_place())
{
    new(p) widget(wd);
    pair<set<widget>::iterator, bool> p = s.emplace(p);
}
```

This seems rather awkward and complicated, and potentially confusing and dangerous. Furthermore, a tricky maneuver would probably be required by the implementation to convert the widget pointer to a node pointer. Worst of all, if the insertion done by `emplace` fails, it will have to quietly destruct the widget you just constructed, leaving p pointing to nothing.

Functor Placement Insert

The `set` situation can be improved considerably by using `bind` and defining a functor that calls `new`. The programmer would write a functor which calls placement `new` with whatever arguments are required. This would look something like:

```
inline void func(void* p, widget_data&& wd)
{
   new(p) widget(forward<widget_data>(wd));
}

set<widget> s;
widget_data wd;     // The data source gets loaded somehow.
pair<set<widget>::iterator, bool> p = s.emplace(bind(func, _1, wd));
```

This solves the problems with set, and it means that for map the emplace function can now return the same type as insert. However, the burden still rests on the user to correctly call `new`, and using functors and `bind` involves more code and a more elaborate syntax than the simple method.

Variadic Placement Insert

Variadic templates allow us to eliminate these remaining problems, yielding a perfect solution. In the case of map for instance, the `emplace` function is called with a `key_type` and the desired constructor arguments. The object is then placement `new` constructed with the -constructor arguments. Using this approach we get:

```
map<long, widget> m;
long id;            // The key gets set somehow.
widget_data wd;     // The data source gets loaded somehow.
pair<map<long, widget>::iterator, bool> p = m.emplace(id, wd);
```

and:

```
set<widget> s;
widget_data wd;     // The data source gets loaded somehow.
pair<set<widget>::iterator, bool> p = s.emplace(wd);
```

To assist in the implementation of `emplace`, and because it's useful in it's own right, I am also proposing a `pair` constructor which takes a parameter pack to construct its second member.

Note that `emplace` can be called with no construction parameters, providing an optimal way to place a default constructed object in a container. Map currently provides this (with `operator[]`), but the others do not.

Container Details

Deque and Vector

Deque and vector require copy operations under various circumstances. However, there is an important class of problems that can be solved by using these containers in ways that do not cause them to move their contents. For this reason I believe that it is worth defining emplacement for them even though the CopyConstructible requirement would remain if they are allowed to resize.

List

List has an assignment, insertion and constructor which take a count *n* and a value to put into each of *n* nodes. Since multiple copies of the type are needed anyway, and copy construction is typically not more expensive than initial construction (often cheaper), I believe the value of placement versions of these is limited. However, Alisdair Meredith suggested that these might in fact be useful. I am going to solicit more feedback on this point.

Map

I have not addressed the possibility of providing variable constructor argument emplacement behavior for the key of a map. It seems to me that the likelihood of this being useful would not outweigh the difficulty of designing an interface that would support it.

Set

There is an implementation consideration for sets. Sets and maps are usually implemented as the same container instantiated on different value types (maps using a pair as the value). But sets need to create their full value type in order to do the comparison (the key *is* the value), so emplace will need to use different code for set and for map.

And what happens if the insert is not successful? The object has been allocated and constructed, so it must now be destroyed. Although this does not have any effect on the container, the construction and destruction of the candidate object might be surprising.

However, there are many cases where failure to insert will not occur (due to program logic) and many others where construction and deletion is not a problem. For this reason I believe that emplacement is valuable for sets.

Names

New names

Several names for these functions come to mind. Possible new names are: `insert_placement`, `placement_insert`, `insert_in_place`, `in_place_insert`, and `emplace`. I like `emplace` because it is short and descriptive, and it received wide approval from the LWG. (Emplace is an English word meaning: *to put in place or position.*)

push_front and push_back

Martin Sebor suggested that it would be easy to provide overloading of these rather than new-name versions. There would be no ambiguities and adding overloads would not break the ABI. I believe there is considerable merit to this because it provides the user with a single "push" that always does the right thing. The interface will be shorter, less confusing, and easier to remember.

The r-value proposal will require that we add an r-value version of push_* anyway, so there is no reason not to add the variadic version instead. The way to keep this from becoming confusing is to redefine push_* to mean calling `emplace` (only). This is always equivalent to or better

than calling `insert`. The effect of calling it with an l- or r-value of the contained object will be exactly the same, and the in-place construction behavior will also be available. For example:

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

// Proposal
template<typename... Args>
void push_back(Args&&... args);

// Example
class widget {
    widget();               // #1
    widget(const widget&);  // #2
    widget(int);            // #3
    widget(float, float);   // #4
};

list<widget> l;

widget w1;
const widget w2;
                                        //     Today        Proposal
l.push_back(w1);                        //     #2           #2
l.push_back(w2);                        //     #2           #2
l.push_back(widget());                  //     #1,#2        #1,#2
l.push_back(widget(42));                //     #3,#2        #3,#2
l.push_back(widget(2.7183, 3.1415));    //     #4,#2        #4,#2
l.push_back(42);                        //     #3,#2        #3
l.push_back();                          //     illegal      #1
l.push_back(2.7183, 3.1415);            //     illegal      #4
```

For these reasons I strongly recommend this approach. In fact, since breaking the ABI is acceptable in this case (and is happening anyway), there is no reason to keep the original push_* functions. I suggest that they be removed entirely.

insert

Overloading `insert` is a possibility that I looked at, and was also suggested by Martin. However, it would create an ambiguity in certain (albeit fairly unusual) cases, and could lead to user confusion. For these reasons I do not recommend this approach.

Other Papers and Issues

N2212

Thorsten Ottosen and I discussed our papers and we concluded that while this proposal covers some of his use cases, the main use case is not adequately addressed. Our proposals will therefore remain separate.

N2069

Thorsten suggested that I may need to use the `decay` type trait to limit instantiations when either the key parameter or constructor parameters are string literals. I am not sure if this is necessary as the situation is not the same as for `make_pair` (in particular, the value pair for

maps is already typed by the time `emplace` is called). I mentioned this to the LWG and there was agreement that this was not an issue.

Library Issue 580 and N2257

Issue 580 contains language that states that containers must construct elements by calling their allocator's `construct` function, so this proposal provids an overloaded `construct` template to handle the in-place construction arguments.

(This raised a significant issue for the LWG, namely that many would like to see `construct` and `destruct` removed from Allocator entirely. At first there was wide consensus was that this was a very good idea, leading to N2257. However, subsequent investigations have shown that this could break existing code, and so the idea has been abandoned.)

Implementation

I originally implemented `emplace` using the Library that ships with Visual Studio 2005. I modified `map` by adding a simulation of the variadic `emplace` signature described above (substituting a single constructor parameter for the parameter pack) and ran both a diagnostic test and a timing test. These tests compared four techniques: naïve `insert`, naïve `operator[]`, my trick using `operator[]`, and `emplace`.

I then implemented `emplace` for map using the latest Concepts GCC compiler with variadic templates and r-value references. I ran only the diagnostic test with that version, however I tested various constructor parameters (including none) to prove that the variadic template solution worked correctly. I also implemented list with both an overloaded and replaced `push_back` and `push_front`, and tested several scenarios.

The diagnostic test produced the following results. "Full" means that the real data has been populated and the copy or destruction has real work to do. Each phase of the test puts one entry into the map, then deletes the map so that the entire life cycle of the contained object is visible. (The `widget_data` class represents the data source required to build the widget.)

```
map<long, widget> m;

m.insert(make_pair(1, widget(widget_data())));

    real data constructor
    copy constructor (full)
    destructor (full)
    copy constructor (full)
    copy constructor (full)
    destructor (full)
    destructor (full)
    destructor (full)

m[1] = widget(widget_data());

    real data constructor
    default constructor
    copy constructor (empty)
    copy constructor (empty)
    destructor (empty)
    destructor (empty)
    copy assignment (full)
    destructor (full)
    destructor (full)

m[1] = widget_data();

    default constructor
    copy constructor (empty)
    copy constructor (empty)
    destructor (empty)
    destructor (empty)
    real data assignment
    destructor (full)

m.emplace(1, widget_data());

    real data constructor
    destructor (full)
```

For the timing test I created a widget that contained enough data (both static and dynamic) to be realistic, then added a large number of them (1,000,000) to a map using each of these methods. The times tended to vary quite a lot from run to run, but the relative performance of the techniques was fairly consistent. Naturally the improvement is highly dependent on the nature of the object—the more expensive the "full" copy, the bigger the gain.

```
    insert              2.0
    op[] = widget       1.8
    op[] = widget_data  1.2
    emplace             1.0
```

Conclusions

Depending on the nature of your object, the performance improvements offered by `emplace` (over the move semantics solution) may be small or large. The limitation that objects in node based containers must be CopyConstructible is unnecessary and surprising. For these reasons I believe that defining emplacement is well worth the effort. It will help me in my work, and will make the Standard Library more complete and useful.

Proposed Wording

General Comments

What follows is wording for all containers and utilities, but it does not address contained element requirements. The CopyConstructible requirement for containers is no longer necessary if the container does not move its objects and `emplace` is used to insert them. I have not tried to specify this now because of the complete change to requirement specifications coming with Concepts. In a future revision of this paper I will provide additional wording for the Concepts-based requirements.

Emplacement operations should become part of the standard container requirements. This is an important fundamental behavior, and code that uses containers needs to be able to depend on it. Designers of future containers (standard or otherwise) will want to support this highly efficient method of insertion.

Requiring emplacement for all containers, relaxing the CopyConstructible requirement, and redefining push_* functions to use emplacement are all transparent to existing source code. No existing code will be broken, although some code will become more efficient.

20.2.3 Pairs [pairs]

Add to paragraph 1, struct pair:

```
template<typename U, typename... Args>
pair(U&& x, Args&&... args);
```

Add after paragraph 4:

```
template<typename U, typename... Args>
pair(U&& x, Args&&... args);
```

Effects: The constructor initializes `first` with `forward<U>(x)` and `second` with `forward<Args>(args)...`

20.6.1 The default allocator [default.allocator]

Remove from class allocator:

```
template <class U>
void construct(pointer p, U&& val);
```

Add to class allocator:

```
template<typename... Args>
void construct(pointer p, Args&&... args);
```

Replace paragraph 12:

```
template <class U > void construct(pointer p , U&& val );
```

12 *Effects:* `::new((void *)p ) T(std::forward<U >(val ))`

With:

```
template<typename... Args>
void construct(pointer p, Args&&... args);
```

12 *Effects:* `::new((void *)p ) T(forward<Args>(args)...)`

23.1 Container requirements

23.1.1 Sequence containers[sequence.reqmts]

Table 88: Sequence container requirements (in addition to container)

Add:

expression	return type	assertion/note pre/post-condition
`template<typename... Args> a.emplace(p, Args&&... args)`	`iterator`	`inserts a T constructed with forward<Args>(args)... before p`

Table 89: Optional sequence container operations

Remove:

expression	return type	assertion/note pre/post-condition	container
`a.push_front(t)`	`void`	`a.insert(a.begin(),t)` Requires:T shall be CopyConstructible.	`list, deque`
`a.push_front(rv)`	`void`	`a.insert(a.begin(),t)`	`list, deque`
`a.push_back(t)`	`void`	`a.insert(a.end(),t)` Requires:T shall be CopyConstructible.	`vector, list, deque, basic_string`
`a.push_back(rv)`	`void`	`a.insert(a.end(),t)`	`vector, list, deque, basic_string`

Add:

expression	return type	assertion/note pre/post-condition	container
`template<typename... Args> a.push_front(Args&&... args)`	`void`	`a.emplace(a.begin(), forward<Args>(args)...)`	`list, deque`
`template<typename... Args> a.push_back(Args&&... args)`	`void`	`a.emplace(a.end(), forward<Args>(args)...)`	`vector, list, deque`

23.1.2 Associative containers [associative.reqmts]

Insert into paragraph 7:

r is a valid dereferenceable const_iterator to a,

Table 90: Associative container requirements (in addition to container)

Add:

expression	return type	assertion/note pre/post-condition	complexity
`template<typename... Args> a_uniq.emplace (Args&&... args)`	`pair<iterator, bool>`	Inserts `t` constructed with `forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of `t`.	Logarithmic
`template<typename... Args> a_eq.emplace (Args&&... args)`	`iterator`	Inserts `t` constructed with `forward<Args>(args)...` and returns the iterator pointing to the newly inserted element.	Logarithmic
`template<typename... Args> a.emplace (r, Args&&... args)`	`iterator`	Equivalent to `a.emplace(forward<Args>(args)...)`. Return value is an iterator pointing to the element with the key equivalent to that of `t`. The const_iterator `r` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Logarithmic in general, but amortized constant if `t` is inserted right after `h`.

23.1.3 Unordered associative containers [unord.req]

Table 92: Unordered associative container requirements (in addition to container)

Add:

expression	return type	assertion/note pre/post-condition	complexity
`template<typename... Args> a_uniq.emplace (Args&&... args)`	`pair<iterator, bool>`	Inserts `t` constructed with `forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of `t`.	Average case O(1), worst case O(a_uniq. size()).
`template<typename... Args> a_eq.emplace (Args&&... args)`	`iterator`	Inserts `t` constructed with `forward<Args>(args)...` and returns the iterator pointing to the newly inserted element.	Average case O(1), worst case O(a_eq.size()).
`template<typename... Args> a.emplace (r, Args&&... args)`	`iterator`	Equivalent to `a.emplace(forward<Args>(args)...)`. Return value is an iterator pointing to the element with the key equivalent to that of `t`. The iterator `r` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Average case O(1), worst case O(a.size()).

23.2.2 Class template deque [deque]

Remove from 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);
```

Add to paragraph 2 – class deque:

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

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

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

23.2.2.3 deque modifiers [deque.modifiers]

Add:

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

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

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

23.2.3 Class template list [list]

Remove from 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);
```

Add to paragraph 2 - class list:

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

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

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

23.2.3.3 list modifiers [list.modifiers]

Remove:

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

Add:

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

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

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

23.2.5 Class template vector [vector]

Remove from paragraph 2 - class vector:

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

Add to paragraph 2 - class vector:

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

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

23.2.5.4 vector modifiers [vector.modifiers]

Remove:

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

Add:

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

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

23.3.1 Class template map [map]

Add to paragraph 2 - class map:

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

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

23.3.2 Class template multimap [multimap]

Add to paragraph 2 - class multimap:

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

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

23.3.3 Class template set [set]

Add to paragraph 2 - class set:

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

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

23.3.4 Class template multiset [multiset]

Add to paragraph 2 - class multiset:

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

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

23.4.1 Class template unordered_map [unord.map]

Add to paragraph 3 - class unordered_map:

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

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

23.4.2 Class template unordered_multimap [unord.multimap]

Add to paragraph 3 - class unordered_multimap:

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

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

23.4.3 Class template unordered_set [unord.set]

Add to paragraph 3 - class unordered_set:

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

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

23.4.4 Class template unordered_multiset [unord.multiset]

Add to paragraph 3 - class unordered_multiset:

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

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

Acknowledgements

At the January 2007 ad hoc Library Working Group meeting, participants made time in a tight schedule to hear my ideas on this subject and encouraged me to submit a proposal. I would like to thank them for their support.

Beman Dawes, Douglas Gregor, and Howard Hinnant were each of tremendous help. They reviewed drafts and contributed ideas, and were very generous with their time despite their own busy schedules. I would like to thank them for their help and encouragement, and acknowledge their specific contributions.

Beman pointed out that allowing objects which are not copy constructible to be put into containers is very important in its own right. He also suggested the functor interface for the non-variadic version as a good solution to the problem with sets, and he encouraged me to include sets (I was a bit daunted by the construct-destruct problem mentioned above). Beman also made several editorial suggestions that improved the presentation considerably.

Doug checked all my variadic template ideas, and pointed out that I should use rvalue references to achieve perfect forwarding. He also very generously helped me to get ConceptsGCC running so that I could try this for real.

Howard suggested that move semantics might be a sufficient solution and should be considered. He also pointed out that allocators had to be addressed since there is a proposal that they be required to take over the construction of contained values.

Early in the Oxford meeting Thorsten Ottosen reviewed the first version of this paper and provided several good suggestions. In particular he pointed out some important use cases for vector that would benefit, and encouraged me to add it.

Martin Sebor suggested overloading push_front and push_back rather than creating separate versions. Alisdair Meredith suggested adding placement versions of other functions (an idea I am still researching).

Revision History

Revision 1 - Since N2217

Changed signatures to include the key value type as a template argument (necessary for perfect forwarding).

Added “hint” signatures.

Added push_* changes.

Added language for container requirements and container signatures.

Removed language for specific functions and for the non-variadic solution.

Added a discussion of other related papers and issues.

Added information about the GCC implementation.

Made numerous other editorial changes.

Revision 2 - Since N2268

Fixed a number of substantive typos in the proposed wording.

Added allocator::construct.

Brought wording inline with N2315.

Removed separate key parameter from maps.

Made a few editorial changes to the narrative text.