# Placement Insert for Containers

*Document number:* N2217 = 07-0077
*Date:* 2007-03-12
*Project:* Programming Language C++
*Reference:* N2134 = 06-0204
*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 that do not move their contents: list, deque, the maps, and the sets. The benefits are improved performance without compromising design, and the ability to put non-copy constructible objects into these containers (with the exception of deque). 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 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 constainers 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.

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.

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. I also look at each container and discuss the special cases.

All but the first of these solutions involve a new member function for the affected containers. Several names for this function come to mind. Overloading `insert` is a possibility, but it would create an ambiguity in certain (albeit unusual) cases, and could lead to confusion. Some other likely names are: `insert_placement`, `placement_insert`, `insert_in_place`, `in_place_insert`, and `emplace`. I like `emplace` because it is short and descriptive.

### Move Semantics Alone

Move semantics will allow the copy construction done by `insert` to be replaced by move construction. This would presumably be a natural consequence of adapting the Library to rvalue references and move semantics. This would make my 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. 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 only 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 delete the widget you just constructed at `p`, leaving you with a pointer 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. (The relationship between the functor and the allocator's `construct` function will require more thought if this version is chosen.)

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 defined to take a `key_type` and a parameter pack. The object is then placement `new` constructed with the parameter pack. With 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 implement this in keeping with the new requirement that containers call the allocator's `construct` rather than using `new` directly, additions to `allocator` and `pair` will also be required: an overload of `construct` which takes a parameter pack to construct the value, and a `pair` constructor which takes a parameter pack to construct its second member.

## Container Details

### List

List is quite similar to map, but requires placement versions of `push_front` and `push_back` in addition to `insert`. Perhaps the names `emplace_front` and `emplace_back` would be easiest to remember.

List also 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.

### Set

The placement insert function would be defined for `set` and `multiset`.

One additional issue pertains to sets. What happens if the insert is not successful? The object has been allocated and constructed, so it now must be destroyed. This must happen within `emplace`, so the user does not have any control over it. The definition of insertion for sets does not specify that failure to insert has no effect, but common sense would strongly suggest it. This function clearly violates that by constructing and destructing an object. However, there are many cases where failure to insert will not occur (due to program logic) and many where construction and deletion is not a problem.

### Map

The placement insert function would be defined for `map`, `multimap`, `unordered_map`, and `unordered_multimap`.

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

### Deque

Deque requires copy operations if insertions are done in the middle of the container, but not if they are done at the ends. There is an important class of problems that can be solved by using a deque that only grows from the ends, so I believe that it is worth defining placement insert for deque, even though the copy constructible requirement would remain.

Like list, deque would require `emplace`, `emplace_front` and `emplace_back`.

## Implementation

I implemented this 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`.

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 is 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 assigment (full)
    destructor (full)
    destructor (full)

m[1] = widget_data();

    default constructor
    copy constructor (empty)
    copy constructor (empty)
    destructor (empty)
    destructor (empty)
    real data assigment
    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 contained objects must be copy constructible is unnecessary and surprising (for example, it would be very natural and useful to put streams into containers). I believe that defining placement insert is worth the effort on the strength of these arguments. It would help me in my work, and would make the Standard Library more complete and useful.

I believe that the variadic solution is by far the best option, and is another in a long list of reasons to approve variadic templates. However, if variadic templates are not approved, I believe that the functor solution is still worth doing and should provide close to the same functionality at the expense of greater complexity. I don't like the idea of introducing more than one way to do the same thing, so I recommend using the functor approach for all containers.

## Proposed Wording

### General Comments

What follows is an approximate wording for the last two solutions, for map only. In a future revision of this paper I will provide exact wording for the solution chosen and for all containers. (I have put `emplace` into section 23.3.1.3, but other places are also possible.)

### Functor Placement Insert

#### 23.3.1.3 map operations [map.ops]

```
template<typename Func>
pair<iterator, bool> emplace(key_type&& x, Func func);
```

*Effects:* Attempts to insert a node into the map at key `x`. If the insertion succeeds, memory is allocated for an instance of the mapped type (`T`) and a pointer to that memory is passed to `func`. Otherwise has the same effects as `insert(const value_type& x)`.

*Returns:* Identical to the return type of: `insert(const value_type& x)`

*Complexity:* Logarithmic.

*Note:* The `T*` passed to `func` must be used in a placement `new` expression to complete the construction of the mapped type object.

### Variadic Placement Insert

#### 23.3.1.3 map operations [map.ops]

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

*Effects:* Attempts to insert a node into the map at key `x`. If the insertion succeeds, memory is allocated for an instance of the mapped type (`T`) and the allocator's `construct` is called with a pointer to that memory and `forward<Args>(args)...`. Otherwise has the same effects as `insert(const value_type& x)`.

*Returns:* Identical to the return type of: `insert(const value_type& x)`

*Complexity:* Logarithmic.

#### 20.6.1.1 allocator members [allocator.members]

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

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

#### 20.2.2 Pairs [pairs]

```
template<typename... Args>
pair(T1&& x, Args&&... args);
```

*Effects:* The constructor initializes `first` with `forward<T1>(x)` and `second` with `forward<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.

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 in the future they will be required to take over the construction of contained values.