Allocator cleanup

Matthew Austern (austern@sgi.com) Hans Boehm (boehm@mti.sgi.com)
Nathan Myers (ncm@adder.cantrip.org)

November 5, 1996

Abstract

The description of Allocators in the September '96 WP is inconsistent, incomplete, and unimplementable as written. Some of the flaws can be easily corrected; others are fundamental, traceable to the original STL proposal. Allocators were accepted on the premise that the loose ends could be worked out before document freeze, but they have not been, yet. This paper presents three options for fixing the WP. All options require changes to Clause 20, with consequential changes in Clauses 21 and 23.

1 Present state of the WP

The current Allocator specification suffers from three sets of problems. First, the member typedefs intended to support alternate memory model extensions, are not sufficiently specified, and almost certainly cannot be specified adequately without major loss of intended functionality. Second, the current description of the interface suffers from a variety of minor errors that make it unimplementable as written. Finally, the consequences of the model on Container semantics have not yet been elucidated, rendering the standard container descriptions incomplete and contradictory.

1.1 Allocator Member Typedefs

The Allocator member typedefs `pointer` and `reference` appeared in the original STL proposal, intended for support of alternate memory models. The semantics required of them were unspecified at the time; like so many other loose ends in that very large proposal, it was assumed that they could be dealt with in due time. No such specifications have been forthcoming.

The semantics of the pointer members could, in principle, be listed completely. However, attempts to do so without overconstraining extensions have revealed problems in other clauses of the draft, and have conflicted with assumptions made by those who have implemented those clauses.

The semantics of the reference members are more troubling: because they cannot usefully be defined to be any standard C++ type other than actual references, the requirements on definitions that rely on extensions are hard to state meaningfully (and have not been). Furthermore, any useful list of such requirements would rely on extensions to the template type deduction apparatus, to deduce new type qualifiers. We do not know of any way to describe such extensions as part of a list of required semantics for a typedef.

To see this, consider the template type deduction problem: `swap` has the signature
`template<class T> void swap(T&, T&)`,
so the template type deduction will fail if it is passed arguments of type `Allocator<T>::reference` if they are not actually references. A conversion from some extended reference type to to a real reference would not be used for the type deduction, and C++ does not provide any mechanism for writing `swap` in a more general way.

This problem is the reason, for example, that the WP contains the function `iter_swap`. However, that special-case approach doesn't help with the dozens of other standard library functions whose arguments are const references. It doesn't, for example, provide a reasonable way of writing a function call like
`find(v1.begin(), v1.end(), v2[0])`.
For the reference typedef to work, container writers would be obliged to use explicit casts from `reference` to `T&` in all calls to standard algorithms. This is incompatible with most potential uses.

The good news is that if the member typedefs were well-specified, they would probably not present any insuperable implementation overhead problems; it is impossible to be certain, though, without a specification to study.

1.2 Numerous Errors

The Utilities Issues list (N1000R1) lists numerous small problems with Allocator in Clause 20. Other related problems appear on the Strings and Containers lists. Most, but not all, of these problems are listed with proposed resolutions. Without resolutions, the Draft is inconsistent and unimplementable as written.

1.3 Allocator Instances

The status quo Allocator is described in terms that imply it can be used as a run-time object, with a copy stored in each container. The default allocator, `allocator<T>`, does not use this feature, and exhibits only static behavior. In a careful implementation, such empty instances need not impose any direct overhead on containers, compared to a fully static interface. However, overhead does appear elsewhere.

The problems with allocator instances are in its implications for container semantics: it implies significant complexity in the description of the Container interface, of the standard containers, and of their implementation.

Much of this complexity has not yet been incorporated in the Draft, with the result that a user cannot expect a standard or user-written container to work properly if specialized for a user-written allocator that displays per-instance semantics. Most of the incomplete specification is associated with two-argument container operations, such as `swap`, which are generally described as having "constant" complexity characteristics.

The "constant" constraint is incompatible with any status quo Allocator that relies on per-instance state. Therefore, to complete the description of these two-argument operations, we must decide for each whether it (1) takes linear time, (2) throws an exception, or (3) exhibits undefined behavior, in the event it is passed containers that use unequal Allocator instances. We must decide whether a single policy is sufficient, or whether each case demands a separate decision.

The status quo can be interpreted to imply (2) by default, in most cases, (1) in a few, and effectively (3) in Clause 21. None of these resolutions is clearly correct; we have seen convincing arguments that each is intolerably wrong. Regardless of these arguments, and the contents of the draft, the existing implementations seem to assume (1) in all cases.

Because there seems to be little agreement on what the correct semantics should be, it seems unlikely that the additional specification can be added at the Kona meeting. If they were to be specified, an implementation overhead issue arises: The inline operations necessary to handle these container operations are more complicated, so it is not always possible to optimize such container code well. These operations involve, for case (1) above, extra loops or, for (2), throws, which might either be dead (but might not be successfully eliminated) or not (and thus could easily interfere with inlining).

1.4 Overview

Given enough time, many of these problems could be solved without dramatically changing the status quo design. However, the Draft changes necessary to specify these solutions would be far larger than is currently permitted given the state of the schedule. Furthermore, some of the problems are entirely intractable, and others do not have obviously correct solutions, which would take more detailed committee attention to resolve.

Allocator instances were added on the assumption that their implications would become clear, and the specification completed, before document freeze. They have not, and we believe it is time to retreat to a fallback position: a stable, well-tested subset of the status quo interface.

2 Options

2.1 Option I: Status quo, with minor fixes

As discussed above, it is impossible to solve the problems of alternative reference types. There are three ways to deal with this problem.

1. Eliminate alternative reference types. Allocators will still have typedefs `pointer` and `const_pointer`, but they do not have typedefs `reference` and `const_reference`.
2. Require that a user-defined memory model must also overload every function in the standard library that take a const reference argument.
3. Document the fact that calls to standard library functions will not work unless every lvalue argument is cast to type `T&`. For example, document that `max(v[0], v[1])` is erroneous code.

None of these alternatives is entirely satisfactory, but the first is probably the least unsatisfactory. Implementing this alternative implies changes in Clauses 20, 21, and 23. It isn't clear whether alternative memory models are genuinely useful if they are restricted to using ordinary references.

Solving the problem of underspecified requirements for `Allocator<T>::pointer` and `Allocator<T>::const_pointer` requires formulating a complete and consistent description of operations on those types. It is uncertain how difficult or time-consuming this will be, or how much new text is required.

The problems related to the semantics of conversion from `Allocator<T>::pointer` to `T*` and `void*` can be solved in one of two ways.

1. Specify the semantics precisely. This involves an unknown amount of new text in Clause 20.
2. Eliminate the conversions. This will require rewriting the descriptions of `Allocator<T>::operator new`, `Allocator<T>::construct`, `Allocator<T>::destroy`, the specialized algorithms in §20.4.4, `basic_string` (Clause 21), and possibly other library components.

The problems related to allocator instances can be solved by specifying the semantics of every operation that involves more than one allocator. This requires changes to Clauses 21 and 23. Known issues include assignment, binary operations on strings, `swap`, mutative `list` operations, and a general policy for mutative operations in user-defined containers. Some of these decisions are controversial: there is still disagreement even about what `swap` should mean in the case of two non-equivalent allocators. In the case of `list` operations, choices include

1. Require that `list` perform a copy, and modify the complexity requirements and that semantics of pointer invalidation accordingly.
2. Throw an exception if allocators compare non-equal.
3. List operations involving two non-equivalent allocators result in undefined behavior.

Option I is in one sense the least radical option, in that it might not require any renumbering of WP sections or any wholesale creation or elimination of features. In another sense it is the most radical option, however, because it will require many small changes in Clauses 20, 21, and 23, and because it implies a rather complicated specification whose consistency is uncertain.

Option I might permit users to define non-standard memory models and to use techniques such as persistence and shared memory. The extent to which these techniquies would be permitted, however, depends on the outcome of decisions that haven't yet been made. The requirements on pointer conversions, for example, are likely to be very significant constraints.

2.2 Option II: Simplified allocators

In this option, allocators no longer attempt to encapsulate alternate versions of pointers and references. Additionally, containers use allocator types instead of allocator instances. This option would still permit users to control containers' allocation strategies.

Option II requires a small amount of new text in Clause 20, and deletion of text from Clauses 21 and 23. The necessary WP changes are listed explicitly in the Appendix.

An implementation of simplified allocators already exists, so we can be confident that they are actually implementable and that they do not impose run-time penalties. The specification is simple enough that we can be reasonable confident that it has no hidden inconsistencies.

2.3 Option III: Removal of allocators from the WP

In this option, containers would not be parameterized by allocation strategy. The library would not include any components to support such parametization. This would involve deleting §20.1.4 and §20.4 entirely, and then removing every template parameter, function argument, and member variable that refers to allocators. Text would have to be deleted from Clauses 17, 20, 21, and 23.

An implementation that conforms to Option II also conforms to Option III, so we can be confident that Option III is implementable. This option, however, requires rather radical changes to the WP: it requires modifying the tables in Clause 17, and extensive renumbering of sections in Clause 20.

Although the WP would be simpler under Option III than under either Option I or Option II, Option III does not simplify the task of library implementors. Library implementors will almost certainly design classes for memory allocation even if the standard does not require them: efficient and thread-safe allocation of small objects is complicated enough that it will usually be encapsulated in a class or function.

Since every library implementation will include memory allocation classes or functions, the real difference between Option II and Option III is whether or not to make them part of the public interface that is visible to users. The strongest argument in favor of doing so is that user-defined classes have just as much need for such a scheme as the predefined classes do.

A Detailed WP changes for Option II

Because of the tight time constraints, we have provided a specific list of all WP changes that are necessary if Option II is adopted. These changes close issues 20–032, 20–041, and 20–045.

A.1 Clause 20 changes

Delete the third paragraph of §20.1.5 [Allocator requirements], and delete the second sentence of the second paragraph. Rewrite the second first and second sentences of the first paragraph so that they no longer refers to alternate pointer types. The new version of §20.1.5 is:

> The library describes a standard set of requirements for *allocators*, which are objects that encapsulate memory allocation. All of the Containers (23), as well as Strings (21) are parameterized in terms of allocators.
>
> Table 41 describes the requirements on types manipulated through allocators. Table 42 describes requirements on allocator types.
>
> The template class member `rebind` in the table above is effectively a template typedef: if the name Allocator is bound to `SomeAllocator<T>`, then `Allocator::rebind<U>::other` is the same type as `SomeAllocator<U>`.

Change Table 41 as follows. Delete lines 3–4 and 6–8. Change the **definition** column of line 5 to read "A value of type `T*` obtained by calling `X::allocate`," and the **definition** column of line 10 to read "A value of type `size_t`". Change the **definition** column of line 9 to read "A value of type U*, obtained by calling X::rebind<U>::other::allocate." The new version of Table 41 is:

Variable	Definition
X	An Allocator class
Y	The type X::rebind<U>::other, for some type U
T	Any type
p	A value of type T* obtained by calling X::allocate
u	A value of type U*, obtained by calling Y::allocate for some Y (possibly X).
n	A value of type size_t

Change Table 42 as follows. Delete lines 9–10, 13–19, and 22–23. Modify lines 1–7 so that the nested typedefs are defined to be specific types: pointer, const_pointer, reference, const_reference, value_type, size_type, and difference_type are defined, respectively, to be T*, const T*, T&, const T&, T, size_t, and ptrdiff_t. Modify lines 11–12 and 20–21 so that they no longer refer to allocator instances. The new version of Table 42 is:

expression	return type	assertion/note pre/post-condition
X::pointer	T*	
X::const_pointer	const T*	
X::reference	T&	
X::const_reference	const T&	
X::value_type	T	
X::size_type	size_t	
X::difference_type	ptrdiff_t	
typename X::rebind<U>::other	Y	If U is T, then Y is X. Y::rebind<T>::other is X.
X::allocate(n) X::allocate(n, u)	T*	memory is allocated for n objects of type T but objects are not constructed. allocate may raise an exception of type bad_alloc or of a type derived from bad_alloc.
X::deallocate(p, n)	(not used)	All n T objects in the area pointed to by p must be destroyed prior to this call. n must match the value passed to allocate to obtain this memory.
X::construct(p, t)	(not used)	Effect: new((void*)p) T(t)
X::destroy(p)	(not used)	Effect: p->~T()

Also add a footnote to the description of X::allocate that reads as follows.

> It is intended that X::allocate be an efficient means of allocating a single object of type T, even when sizeof(T) is small. That is, there is no need for a container to maintain its own "free list".

In §20.4 [lib.memory], delete the definitions of `operator new`, `operator delete`, `operator==`, and `operator!=` from the **Header** `<memory>` **synopsis**. That is, delete the six lines that immediately follow the line "`template <> class allocator<void>`".

In §20.4.1 [lib.default.allocator], delete all of the member functions except for `allocate`, `deallocate`, `construct`, and `destroy`, and change the signatures of those member functions so that they are declared as static. Delete the global operators `operator==`, `operator!=`, and `operator new`. After these deletions, the declaration of class `allocator`, and of its void specialization, reads

```
namespace std {
  template <class T> class allocator;

  // specialize for void:
  template <> class allocator<void> {
  public:
    typedef void*       pointer;
    typedef const void* const_pointer;
    // reference-to-void members are impossible.
    typedef void value_type;
    template <class U> struct rebind { typedef allocator<U> other; };
  };

  template <class T> class allocator {
  public:
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T&        reference;
    typedef const T&  const_reference;
    typedef T         value_type;
    template <class U> struct rebind { typedef allocator<U> other; };

    static T* allocate(size_t n, const void* hint = 0);
    static void deallocate(T* p, size_t n);

    static void construct(T* p, const T& val);
    static void destroy(T* p);
  };
}
```

In §20.4.1.1 [lib.allocator.members], delete the definitions of the members `address` (both versions) and `max_size`.

Delete §20.4.1.2 [lib.allocator.globals].

Replace §20.4.1.3 [lib.allocator.example], which will be renumbered as §20.4.1.2 because of the deletion of [lib.allocator.globals], with the following new text.

20.4.1.2 Example allocator **[lib.allocator.example]**

[*Example:* Here is a sample container parameterized on the allocator type.

```
template <class T, class Allocator = allocator<T> >
class AContainer {
  struct Treenode;
  typedef typename Allocator::rebind<Treenode>::other
          Treenode_allocator;
  struct Treenode { Treenode* left_, right_; T t; };
public:
  AContainer() : root_(0) {}
  // ...
private:
  Treenode* root_;
  Treenode* get_node() {
    return new(Treenode_allocator::allocate(1, root_)) Treenode;
  }
};
```

Here is a sample allocator that simply calls `malloc`. It is useful for debugging, and for working with leak-detection and bounds-checking software.

```
template <class T>
class malloc_alloc : public allocator<T> {
  template <class U> struct rebind { typedef malloc_alloc<U> other; };

  static T* allocate(size_t n, const void* = 0) {
    return (T*) std::malloc(n * sizeof(T));
  }
  static void deallocate(T* p, size_t) { std::free(p); }
};
```

–*end example*]

[*Note:* In addition to the default allocator `allocator`, it is recommended, but not required, that implementations also supply the allocators `gc_allocator` and `fast_allocator`.

The class `gc_allocator` allocates garbage-collectable memory. Access to memory allocated using `gc_allocator` after the memory has become unreachable is undefined. `gc_allocator::deallocate` has no effect.

The template `fast_allocator<>` works identically to the default, `allocator<>`, in single-threaded environments. However, it is not necessarily safe to call its member functions concurrently in multi-threaded environments. In such environments, `fast_allocator<>` can often be implemented much more efficiently than can the default allocator.

–*End Note*]

A.2 Clause 21 changes

- In §21.3 [lib.basic.string], remove the allocator argument from all of `basic_string`'s constructors in which it appears, and remove the `explicit` declaration from the default constructor. Also remove the member function `get_allocator` from the "*// string operations*" section.
- In §21.3.1 [lib.string.cons], remove the Allocator argument from all of `basic_string`'s constructors and from the headings in all of the tables that describe constructor semantics.

Remove the `explicit` declaration from `basic_string`'s default constructor. Delete paragraph 1 of §21.3.1 [lib.string.cons].

- In §21.3.5.8 [lib.string::swap], change the **Complexity** clause (paragraph 3) to read "constant time".
- In §21.3.6 [lib.string.ops], delete paragraph 6, and the member function declaration (of `get_allocator`) that appears immediately above it.

A.3 Clause 23 changes

- In Table 75 (which is in §23.1 [lib.container.requirements]) delete the line that defines the expression `a.get_allocator()`. In the line that defines the expression `a.swap()`, change the entry in the complexity column to "constant time".
- Delete paragraph 8 of §23.1 [lib.container.requirements].
- In the declarations of `deque`, `list`, `vector`, and `vector<bool>`, in, respectively, §23.2.1 [lib.deque], §23.2.2 [lib.list], §23.2.4 [lib.vector], and §23.2.5 [lib.vector.bool], remove the Allocator argument from the three constructors in which it appears. Remove the declaration `explicit` from the default constructor (which now takes no arguments). Remove the member function `get_allocator`.
- In each of §23.2.1.1 [lib.deque.cons], §23.2.2.1 [lib.list.cons], and §23.2.4.1 [lib.vector.cons], remove the Allocator argument from each constructor where it appears, and remove the declaration `explicit` from the default constructor. Delete the phrase "using the specified allocator" from the descriptions of the constructors.
- In the declarations of `queue`, `priority_queue`, and `stack`, in, respectively, §23.2.3.1 [lib.queue], §23.2.3.2 [lib.priority.queue], and §23.2.3.3 [lib.stack], remove the `get_allocator` member function. Remove the Allocator argument from the constructor. In §23.2.3.1 [lib.queue] and §23.2.3.3 [lib.stack], but not in §23.2.3.2 [lib.priority.queue], also remove the `explicit` declaration from the constructor.
- In the declarations of `map`, `multimap`, `set`, and `multiset`, in, respectively, §23.3.1 [lib.map], §23.3.2 [lib.multimap], §23.3.3 [lib.set], and §23.3.4 [lib.multiset], remove the Allocator argument from the two constructors in which it appears. Remove the declaration of the member function `get_allocator`.
- In §23.3.1.1 [lib.map.cons], §23.3.2.1 [lib.multimap.cons], §23.3.3.1 [lib.set.cons], and §23.3.4.1 [lib.multiset.cons], remove the Allocator argument from the two constructors where it appears, and remove the phrase "and allocator" from paragraphs 1 and 3.

B Acknowledgments

We wish to thank Alex Stepanov and Andy Koenig for helpful suggestions, and Bjarne Stroustrup for bringing some allocator issues to our attention.